USC UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

CURSO 2022/2023

MEMORIA
APARCAMIENTO

GUILLERMO ARCOS SALGADO,
CAROLINA ALBA GARCÍA,
LUÍS BARBA CEPEDELLO,
DANIEL PIQUERAS VALLE,
SERGIO LAGO BESADA,
CELTIA CASTELO GAJINO

1. Introducción y objetivos

Objetivos iniciales

Trabajando sobre el proyecto original *Proyecto 2F Gestión de un Aparcamiento de la USC*, se busca una mejora de este a través de un análisis y rediseño; corrigiendo el modelo y añadiendo pequeñas mejoras. El objetivo principal de mejora es desarrollar una aplicación con unas ciertas funcionalidades que permitan conectarse con la base de datos y realizar diferentes transacciones para leer, insertar o eliminar datos.

En nuestro caso particular, vamos a desarrollar una aplicación en Java para modelar una red de aparcamientos de la USC, que estará destinada a los trabajadores del aparcamiento. Debe permitir crear usuarios, reservar plazas a los usuarios y reflejar las plazas en las que aparcan, así como la fecha. Además, debe calcular el precio que tiene que pagar cada usuario por reservar o aparcar en una plaza. Por último, también tiene que guardar registro de los usuarios que cometen infracciones y son vetados, no dejándoles usar los aparcamientos, y calcular varias estadísticas que pueden resultar de interés.

Plan de proyecto

- Corrección del modelo anterior

Para empezar, analizaremos la semántica y añadiremos pequeñas modificaciones para ampliar la funcionalidad. Por ejemplo, en vez de modelar las plazas de un solo aparcamiento, modelaremos una red de aparcamientos.

En cuanto al Modelo Entidad-Relación y el Modelo Relacional, tomaremos los cambios que hemos realizado en la semántica y los reflejaremos en estos. De esta forma, tendremos un proyecto completo sobre el que podremos trabajar.

Por último, volveremos a generar los scripts de la base de datos implementando todos los cambios.

En la memoria, presentaremos el nuevo diseño del proyecto y añadiremos una sección para señalar las modificaciones realizadas sobre el proyecto inicial.

- Determinación de las funcionalidades e implementación de la aplicación.

Una vez rediseñado el modelo anterior, elegiremos las funcionalidades a implementar en la aplicación, teniendo en cuenta que resulten de interés para los trabajadores del aparcamiento.

Para terminar, pasaremos a crear la aplicación en Java usando la biblioteca gráfica Swing. Para establecer la conexión con la base de datos, creada en PostGreSQL, se usará la API de JDBC.

2. Análisis Semántico

La Universidad de Santiago de Compostela desea tener una base de datos para la organización de sus aparcamientos. Esta debe contar con información de usuarios, vehículos, plazas y aparcamientos. Para esto se exponen las consideraciones siguientes, a tener en cuenta para su diseño.

La USC cuenta con varios aparcamientos repartidos por el campus, cada uno de ellos identificado con un id único. También se quiere conocer su dirección y su aforo (esto es, el número de plazas con las que cuenta).

De las personas que estacionen su vehículo en uno de estos aparcamientos, se desea conocer su DNI, nombre completo, teléfono de contacto y correo electrónico. Si son personal de la USC, se quiere almacenar la fecha de ingreso en esta institución y el rol (personal de administración y servicios, alumno o personal docente) que desempeñan en la misma. Aunque alguien de la USC abandone la institución, se le seguirá "tratando" como si todavía lo estuviera, como agradecimiento por su labor en la Universidad.

Cada persona puede poseer varios vehículos, de los cuales interesa su matrícula, el tipo de vehículo (moto, coche o vehículo grande), marca, modelo y año de matriculación. Un vehículo siempre tiene un único propietario.

En los aparcamientos hay dos tipos de plazas: las destinadas a que cualquier coche aparque en ellas y las destinadas a ser reservadas por los usuarios por un cierto período de tiempo (durante el cual solo ellos las podrán ocupar). Por motivos de seguridad/mantenimiento, no se podrá nunca aparcar en una plaza de las que están disponibles para reservar ni, claro está, reservar una de las del tipo "aparcar".

Las plazas para reservar/aparcar se identifican por un código numérico único y pueden ser, a su vez, de tres subtipos: C(para coches), M (para motos) o G (para vehículos grandes). Además, se exige que solamente se permita ocupar la plaza con un vehículo del tipo correspondiente.

Se necesita mantener un registro detallado de la actividad del parking: en qué plazas se aparcó/fueron reservadas, por quién (y con qué coche), la fecha de entrada, de salida y el precio pagado (los detalles del cálculo del precio se especifican a continuación).

Los pagos se realizan a través de una nueva aplicación de la USC que vincula la cuenta bancaria de cada cliente con la matrícula de sus vehículos. Los miembros de dicha institución pagarán una cantidad fija (50 cts) por hora al aparcar y podrán reservar por cierto tiempo - a 7,5 euros por día - una plaza de las disponibles del tipo "reservar".
Cabe destacar que las reservas de plaza siempre comenzarán a las 9 de la mañana de la fecha actual y terminarán a las nueve de la mañana del día seleccionado.

Las personas ajenas a la USC tendrán que pagar 1 euro por hora aparcada y no podrán tener sus vehículos aparcados más de 72 horas seguidas, ni hacer una reserva de más de 7 días. Tanto para los integrantes de la USC como los que no, se intentará cobrar el importe correspondiente cuando se salga del parking o cuando termine la fecha de reserva.

La USC cuenta también con un sistema de infracciones. Se considerará una infracción que los conductores ajenos a la USC no respeten el límite de 72 horas. Retrasarse en el pago (por no tener fondos) o realizar maniobras imprudentes son otros tipos de infracciones (aplicables también al personal de la USC). Si una persona comete 5 infracciones no podrá volver usar el parking para estacionar su/s vehículo/s, siendo relevante guardar la fecha en la que fue vetado. De los no-vetados se desea saber cuántas infracciones han cometido.

3. Modelo Conceptual

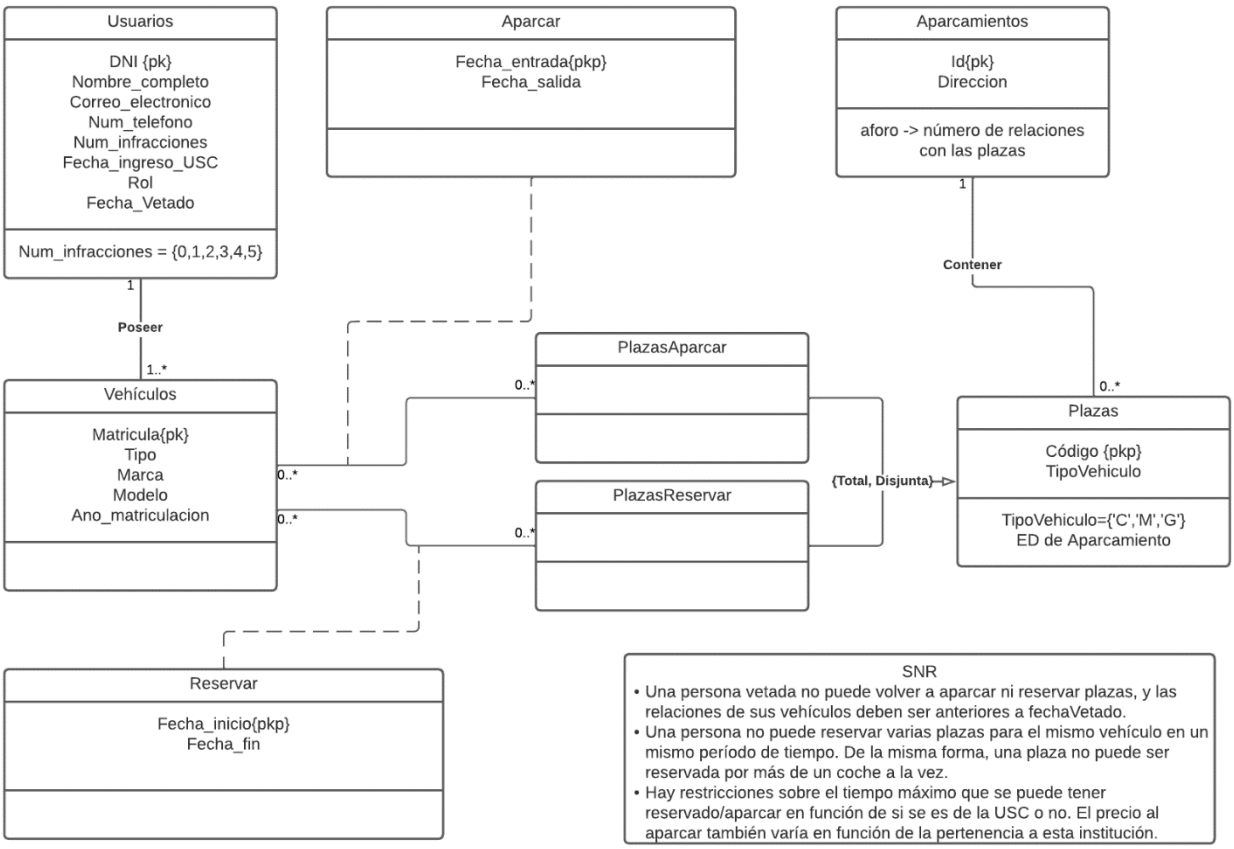
A partir de la semántica podemos definir las siguientes entidades: los usuarios del sistema de aparcamientos (**Usuarios**), los diferentes vehículos que poseen los usuarios (**Vehículos**), los aparcamientos existentes (**Aparcamientos**) y las plazas disponibles en cada uno de ellos (**Plazas**). Esta última considerada entidad débil, pues depende existencialmente de Aparcamientos: cada aparcamiento puede (**Contener**) diversas Plazas (1:N), que cuentan con un código identificador como clave primaria parcial para distinguir las diferentes plazas de un aparcamiento.

Por otra parte, se ha decidido separar Plazas en una jerarquía total disjunta: por un lado, las que están destinadas para aparcar (**PlazasAparcar**), y por otro las que están destinadas para reservar (**PlazasReservar**).

Estas son las entidades entre las que se establecen las relaciones que describe la semántica. En primer lugar, cada Usuario puede (**Poseer**) Vehículos, y cada Vehículo es poseído por un único Usuario (relación 1:N).

Por otra parte, se establece una relación (**Aparcar**) entre Vehículos y PlazasAparcar, donde se guarda la fecha de entrada al aparcamiento, la fecha de salida. Además, tenemos una relación N:N donde sería posible que un vehículo aparque en la misma plaza más de una vez; por tanto, elegimos la fecha de entrada como clave primaria parcial de la relación para diferenciar dichas relaciones. Lo mismo sucede con Vehículos y PlazasReservar donde colocamos la relación (**Reservar**) con los mismos atributos.

Modelo Entidad-Relación



Modelo Entidad-Relación

Diccionario de Datos

Entidad	Descripción	Número de Instancias Previsto
Usuarios	Datos de los usuarios del sistema de aparcamiento	Número de usuarios que hacen uso del sistema
Vehículos	Datos de los vehículos registrados	Número total de vehículos que han sido registrado
PlazasAparcar	Datos de las plazas disponibles para aparcar	Número de plazas reservadas para aparcar
PlazasReservar	Datos de las plazas disponibles para reservar	Número de plazas reservadas para reservar
Plazas	Datos de las plazas existentes	Número total de plazas que existen
Aparcamientos	Datos de los distintos aparcamientos de la USC	Número total de aparcamientos que existen

Entidad	Multiplicidad	Relación	Multiplicidad	Entidad
Usuarios	1	Poseer	1…*	Vehículos
Plazas	0…*	Localizarse en	1	Aparcamientos
Usuario	0..*	Reservar	0..*	PlazasReservar
Vehículo	0..*	Reservar	0..*	PlazasReservar
Usuario	0..*	Aparcar	0..*	PlazasAparcar
Vehículo	0..*	Aparcar	0..*	PlazasAparcar

Entidad o Relación	Atributos	Descripción	Tipos de Datos	N	M	D	P
Usuario	dni{PK}	DNI del usuario	carácter 9	no	no	no	no
	nombre	Nombre y apellidos del usuario	carácter variable 500	no	no	no	no
	telefono	Número de teléfono del usuario	carácter 9	si	no	no	no
	correo	Correo electrónico del usuario	carácter variable 500	si	no	no	no
	fechaIngresoUSC	Fecha de ingreso en la USC	date	si	no	no	no
	rol	Puesto que ocupa en la universidad	carácter variable 500 (administración, docente, alumno, noUSC)	no	no	no	no
	fechaVeto	Fecha de veto del sistema de aparcamiento	date	si	no	no	no
	numeroInfracciones	Numero de infraccione cometidas por el usuario	número (0,1,2,3,4,5)	no	no	no	0
Vehículo	matrícula{PK}	Matrícula del vehículo	carácter 7	no	no	no	no
	tipo	Letra que clasifica al vehículo según sea coche, moto o vehículo grande	carácter 1 (C,M.G)	no	no	no	no
	marca	Marca del vehículo	carácter variable 500	si	no	no	no
	modelo	Modelo del vehículo	carácter variable 500	si	no	no	no
	anhoMatriculación	Año en que se matriculó el vehículo	numérico(4,0)	si	no	no	no
Aparcamiento	id{PK}	Código único de cada aparcamiento	carácter 5	no	no	no	no
	dirección	Dirección en que se encuentra el aparcamiento	carácter variable 1000	si	no	no	no
	Aforo	Número de plazas del aparcammiento	numérico (3,0)	no	no	no	no

Plazas	codigo {PK}	Código único que identifica cada plaza	numérico (2,0)	no	no	no	no
	tipo	Letra que indica para qué vehículo (coche, moto o vehículo grande) es la plaza.	carácter 1 (C,M.G)	no	no	no	no
Aparcar	fechaEntrada{PK}	Fecha y hora en la que se aparca en la plaza	timestamp	no	no	no	no
	fechaSalida	Fecha y hora en la que se abandona la plaza	timestamp	si	no	no	no
Reservar	fechaInicio{PK}	Fecha y hora en que comienza la reserva de la plaza	timestamp	no	no	no	no
	fechaFin	Fecha y hora en que termina la reserva de la plaza	timestamp	si	no	no	no
Poseer							
Localizarse en							

N: Admite Nulos; M: Multivaluado; D: Derivado; P: Predeterminado

4. Modelo Lógico

Explicación

Para pasar el modelo Entidad-Relación al modelo Relacional, debemos aplicar una serie de normas. Comenzaremos con la transformación del conjunto de entidades:

Primero, las entidades del modelo E-R deben ser claramente identificadas. Analizaremos primero las que no forman parte de la jerarquía, las que no son subclase ni superclase de ninguna otra clase del modelo. Se pueden distinguir: Usuarios, Vehículos y Aparcamientos. Cada una de estas entidades generará una nueva tabla, cuyo nombre será el mismo que el empleado en el modelo E-R y se mantendrán los atributos según lo descrito en el modelo entidad relación al no aparecer ninguno de tipo compuesto (que habría que separar en atómicos) o multivalorado (que generaría otra tabla). Además, todas estas entidades fuertes mantendrán la clave primaria del modelo E-R.

La entidad Plazas es una superclase de las entidades PlazasAparcar y PlazasReservar, formando las tres una jerarquía total disjunta. Por tanto al pasar al modelo relacional aparecerán dos tablas representando a las subclases (no aparecerá una tabla Plazas), ambas con los atributos de la superclase únicamente al carecer de atributos propios. De este modo, se crean las tablas PlazasAparcar y PlazasReservar.

Al ser ambas entidad débil de Aparcamientos, su clave primaria será la dupla formada por la copia de la clave primaria de Aparcamientos (*idAparcamiento*) y su clave primaria parcial (*código*). Además, *idAparcamiento* también será clave externa, referenciando en la tabla Aparcamientos al id del aparcamiento; asegurando así que este atributo solamente pueda tomar valores que hayan sido tomados previamente en la tabla referenciada.

A continuación, llevaremos a cabo la transformación de los conjuntos de relaciones.

En las relaciones 1:N, no se generan nuevas tablas, sino que se incluye una copia de la clave primaria del lado 1 en la tabla del lado N (además de los atributos de la relación). Ese nuevo atributo incluido en la tabla del lado N será una clave externa que referencia a la clave primaria en la tabla de lado 1. Nos encontramos con dos relaciones de este tipo:

1) En la relación Poseer, la clave primaria de Usuarios (DNI) se copia y se incluye en la tabla Vehículos con el nombre *dni* donde figurará como clave foránea.
2) En la relación Localizarse en, entre Aparcamientos y Plazas, también se incluiría la copia de la clave de Aparacamientos en Plazas; pero ya se realizó en la etapa anterior por ser Plazas entidad débil de Aparacamientos.

Con respecto a las relaciones binarias N:N, se genera una tabla con copias de las claves primarias de las entidades y los atributos de la relación. La clave primaria estará formada por la clave primaria parcial de la relación (si existe) y las copias de las claves primarias de las entidades, que además serán claves externas que referencien a las claves primarias de sus respectivas tablas Nos encontramos de nuevo con dos relaciones de este tipo:

1) Se genera la tabla Aparcar, en la que se incluye como clave foránea una copia de las claves de Vehículos (*matriculaVehículo* referencia *matricula* en la tabla Vehículos) y PlazasAparcar (la dupla *codigoPlaza-idAparcamiento* referencia la dupla *codigo-idAparcamiento* en la tabla PlazaAparcar, y esta última referencia a su vez *id* en la tabla Aparcamientos). Además, también aparecen los atributos propios de la relación que son *fechaEntrada* y *fechaSalida*. La clave primaria la conformarán 4 atributos: las referencias a Vehículos y PlazasAparcar (*matriculaVehículo, codigoPlaza* e *idAparcamiento*), y el atributo *fechaEntrada*.

2) Análogamente se genera la tabla Reservar, en la que se incluye como clave foránea una copia de las claves de Vehículos (*matriculaVehículo* referencia *matricula* en la tabla Vehículos) y PlazasReservar (la dupla *codigoPlazaReserva-idAparcamiento* referencia la dupla *codigo-idAparcamiento* en la tabla PlazaReservar, y esta última referencia a su vez *id* en la tabla Aparcamientos). Además, también aparecen los atributos propios de la relación que son *fechaInicio* y *fechaFin*. La clave primaria la conformarán otra vez 4 atributos: las referencias a Vehículos y PlazasReservar (*matriculaVehículo, codigoPlazaReserva* e *idAparcamiento*), y el atributo *fechaInicio*.

Modelo Relacional

Usuarios (dni, nombre, teléfono, correo, fechaIngresoUSC, rol, fechaVeto, numeroInfracciones)
PK: dni
Forma Normal: BC

Vehiculos (matricula, dni, tipo, marca, modelo, anhoMatriculacion)
PK: matricula
Forma Normal: BC
FK: dni REFERENCIA Usuarios (dni)
Borrado: Set Null
Actualización: Cascade

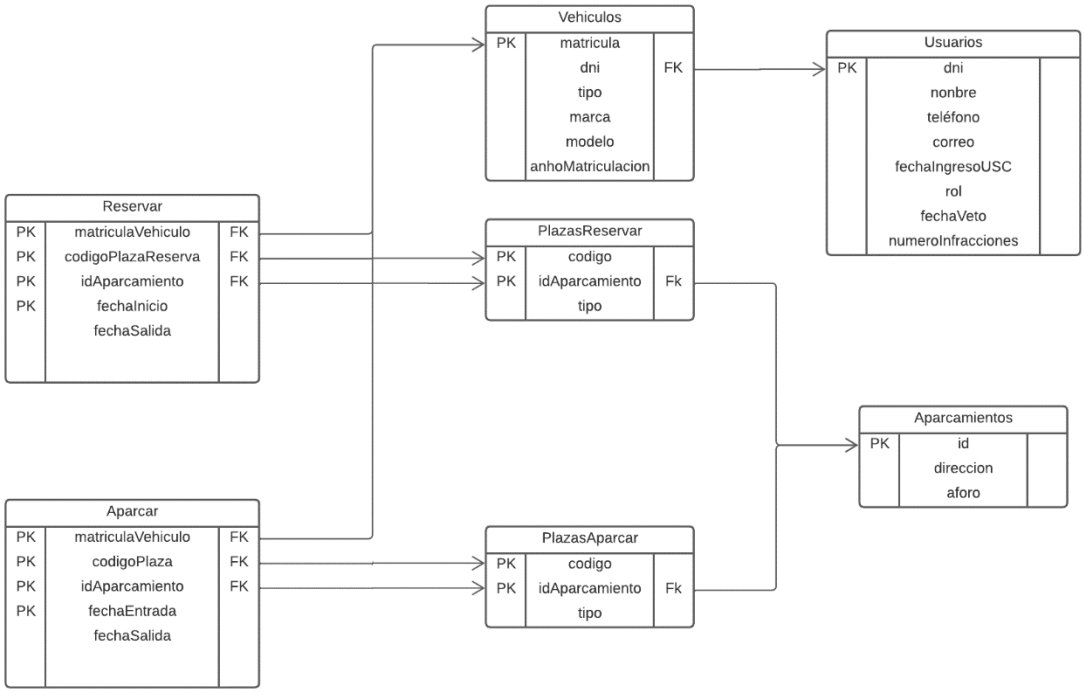
Aparcamientos(id, direccion, aforo)
PK: id
ATRIBUTO CALCULADO: aforo
Forma Normal: BC

PlazasReserva (codigo, idAparcamiento, tipo)
PK: codigo, idAparcamiento
FK: idAparcamiento REFERENCIA Aparcamientos(id)
Borrado: No Action
Actualización: Cascade
Forma Normal: BC

PlazasAparcar (codigo, idAparcamiento, tipo)
PK: codigo, idAparcamiento
FK: idAparcamiento REFERENCIA Aparcamientos(id)
Borrado: No Action
Actualización: Cascade
Forma Normal: BC

Aparcar(matriculaVehiculo, codigPlaza, idAparcamiento, fechaEntrada, fechaSalida)
PK: matriculaVehiculo, codigoPlaza, idAparcamiento, fechaEntrada
Forma Normal: BC
FK: matriculaVehiculo REFERENCIA Vehiculos(matricula)
Borrado: Set Null
Actualización: Cascade
FK: codigoPlaza, idAparcamiento REFERENCIA PlazasAparcar(codigo, idAparcamiento)
Borrado: Set Null
Actualización: Cascade

Reservar (matriculaVehiculo, codigoPlazaReserva, idAparcamiento, fechaInicio, fechaFin)
PK: matriculaVehiculo, codigoPlazaReserva, idAparcamiento, fechaInicio
Forma Normal: BC
FK: matriculaVehiculo REFERENCIA Vehiculos(matricula)
Borrado: Set Null
Actualización: Cascade
FK: codigoPlazaReserva, idAparcamiento REFERENCIA PlazasReserva(codigo, idAparcamiento)
Borrado: Set Null
Actualización: Cascade



Modelo relacional

Políticas de Integridad Referencial

En vista de la utilización de claves externas, es necesario definir las políticas de integridad referencial aplicadas con el fin de mantener la consistencia en nuestra base de datos.

Cuando un usuario se quiera dar de baja, por la 'Ley de protección de datos' estamos obligados a eliminar su DNI de nuestra base de datos. Sin embargo, con el objetivo de mantener un libro de cuentas con los ingresos derivados del aparcamiento, a la USC no le interesa que se pierdan los datos relacionados con ese usuario. Por ello, establecemos en la tabla de Vehículos una política de *Set Null* que ponga a null el DNI del *usuario*, manteniendo así el resto de información de las tuplas.

Para el borrado del idAparcamiento en la tabla Plazas, se ha elegido una política de *No Action*. De esta forma, si se intenta borrar un aparcamiento, pero aún queda alguna plaza en el mismo, se producirá un error que no permitirá borrarlo. Solo se podrá borrar un aparcamiento si este no tiene plazas.

En la tabla Aparcar, de cara a mantener un registro histórico, resulta muy importante no perder información de estas tuplas. Ya sea para estudiar la tendencia de la comunidad a utilizar un tipo de vehículo u otro, el porcentaje de personas externas que lo utilizan con el paso del tiempo... Es por ello que elegimos una política de *Set Default*, creando las siguientes tuplas ficticias que nos servirán para referenciar a ellas las claves foráneas de Aparcar cuando estas sean eliminadas en sus tablas principales:

-Vehículo: ('0000AAA', 'C', null, null, null, null)

-Aparcamiento: ('PARK0', null, 2)

-PlazaAparcar: (0,'C','PARK0')

Por tanto, cuando se elimine alguna de las tuplas de Aparcar, automáticamente los atributos *codigoPlaza* e *idAparcamiento* pasarán a referenciar a estos datos fictios; cambiando dichos atributos a por *0* y *PARK0*. Así tendremos información de la duración de un aparcamiento sin importar en qué plaza concreta se realizó. De la misma forma establecemos el borrado de la clave que referencia a la matrícula en la tabla Vehículos cambiándola por *0000AAA*, ya que además aquí sí que estamos obligados a eliminarla por la 'Ley de protección de datos'.

Por último, en la tabla Reservar establecemos las mismas políticas al ser completamente análogas. Establecemos por tanto *Set Default* en sus tres claves externas creando la siguiente tupla ficticia a mayores de las anteriores:

-PlazaReservar: (0,'C','PARK0')

Cabe mencionar que, con el fin de mantener la consistencia de nuestra base de datos, todas las actualizaciones las realizamos según el método *cascade*. De esta forma si se actualiza un valor en el atributo referenciado de la tabla principal, los valores que referencian ese atributo se actualizarán automáticamente.

5. Cambios sobre el modelo original

Se resumen a continuación los principales cambios con respecto al proyecto original.

Semántica

- En vez de modelar un solo aparcamiento, modelaremos una red de aparcamientos. Para cada uno, queremos registrar el id, la dirección y el aforo.

Modelo Entidad-Relación

- Se ha añadido la entidad Aparcamientos.
- Se han cambiado las agregaciones Aparcar y Reservar por relaciones N:N Vehículo-PlazaAparcar y Vehículo-PlazaReserva: no consideramos necesaria la agregación entre Poseer y PlazasAparcar porque, dado que un vehículo solo puede estar asignado a un único usuario, es suficiente con relacionar Vehículo con PlazasAparcar y con PlazasReservar.
- Se ha añadido una jerarquía total disjunta para las plazas y se ha añadido la entidad débil Plazas: antes no era necesaria porque la superclase Plazas no se relacionaría con ninguna otra entidad. No obstante, como hemos añadido Aparcamientos y la relación LocalizarseEn ahora resulta más sencillo crear dicha superclase y relacionarla con Aparcamientos, en vez de tener dos relaciones diferentes para LocalizarseEn.
- Se ha añadido la condición de que una plaza no puede ser reservada por más de una persona a la vez.
- Se ha eliminado la SNR referente a la agregación.
- Se han eliminado los atributos calculados tiempo y precio de Reservar y Aparcar. Ahora se calculan únicamente en la aplicación Java.

Modelo Relacional

Como consecuencia de las modificaciones introducidas en la semántica y en el modelo Entidad-Relación, se observan también varios cambios en el modelo relacional:

- Se han modificado los nombres de algunos atributos para simplificarlos.
- Se han cambiado varias de las políticas de integridad referencial relacionadas con el borrado en las tablas *Aparcar* y *Reservar*, de Cascada a Set Default.
- La jerarquía disjunta-total no modifica las tablas, porque sigue generando las tablas *PlazaAparcar* y *PlazaReserva*(una por cada combinación superclase-subclase).
- Se ha añadido la tabla *Aparcamientos*.
- Se ha añadido el atributo *idAparcamiento* como clave foránea a las tablas *PlazaReserva* y *PlazaAparcar*, a raíz de la relación *Conte*. Además, es también clave primaria porque son Entidades Débiles de *Aparcamiento*.

6. Aspectos generales de la Aplicación

6.1 Introducción

Para implementar todo lo expuesto anteriormente y añadir nuevas funcionalidades que serán tratadas en la siguiente sección, hemos creado una aplicación de Java que se conecta a una base de datos **PostgreSQL** utilizando **JDBC** (API que permite acceder y manipular bases de datos).

Por otra parte, hemos utilizado la biblioteca **Swing** de Java para crear la interfaz de usuario de nuestra aplicación. Se trata de una biblioteca de componentes gráficos de Java que proporciona una amplia variedad de elementos de interfaz, como botones, campos de texto, etiquetas, tablas, etc.

En cuanto a la estructura de nuestro proyecto, está dividido en tres paquetes diferentes:

- ***Aplication:*** contiene clases de Java equivalentes a las tablas creadas en SQL (**Usuario**, **Vehiculo**, **PlazaAparcar**...), lo que permite crear objetos de Java que representen los datos de la tabla. Además, se han añadido **RolUsuario** y **TipoPlaza**, que son enumeraciones y que nos servirán para restringir los valores que pueden tomar esos datos. Contiene también la clase **FachadaAplicación**, donde se localiza el main de la aplicación. Es desde esta clase desde donde se inicializa la vista, llamando a clases contenidas en los otros dos paquetes para crear la conexión con la base de datos e inicializar la interfaz.

- ***ConexionBBDD*:** contiene la clase **FachadaBBDD**, desde donde se crea la conexión con la base. Para cargar la configuración de la base en Java y realizar la conexión, leemos los detalles de configuración desde un archivo de propiedades llamado *"baseDatos.properties*". También se encuentran en este paquete **DAOUsuarios** y **DAOPlazas**, que heredan de **AbstractDAO**. Este patrón, utilizado normalmente para separar la lógica de acceso a los datos de la lógica de negocio y la presentación, lo hemos empleado con el objetivo de separar responsabilidades y diferenciar las funciones relacionadas con plazas y las relacionadas con usuarios y sus vehículos.

- ***Gui***: es en este paquete donde se encuentra todo lo relacionado con Swing y la interfaz gráfica. Encontramos diferentes modelos de tablas, utilizados para mostrar datos de la base en JTables; múltiples JDialogs, en los que se llevarán a cabo acciones relativas a las funcionalidades y la clase **FachadaGui**, desde donde se inicializa la vista llamando a la ventana principal y a la ventana de autentificación de usuario (Ir a Funcionalidad 8).

6.2 Bibliotecas de Java: PostgreSQL, JCalendar, JTattoo

Se exponen a continuación las bibliotecas utilizadas:

- PostgreSQL: una biblioteca de software libre y de código abierto que se utiliza para conectarse a una base de datos PostgreSQL desde aplicaciones Java.

- JCalendar: una biblioteca de componentes de interfaz de usuario para Java Swing que proporciona un calendario gráfico para seleccionar fechas en aplicaciones Java.
- JTattoo: una biblioteca de componentes de interfaz de usuario para Java Swing que permite cambiar el estilo de las ventanas y los botones.

Cabe destacar que la aplicación puede no compilar o bien, ejecutarse con errores, si al cargar el proyecto no se configuran estas dos bibliotecas correctamente.

6.3 Gestión de transacciones (begin, commit, serializable)

Con la intención de introducir conceptos del temario de Bases de Datos II, se propone un sistema de control de concurrencia mediante la imposición de ciertos niveles de aislamiento que evitarán conflictos a nivel de transacción. En este caso, mostramos un ejemplo con dos funcionalidades íntimamente relacionadas que conllevan transacciones que pueden entrar en conflicto: la funcionalidad de reservar o aparcar plazas y la funcionalidad de eliminación de plazas. Para ello procederemos como en el caso de las prácticas del tema 2.

En primer lugar, describamos el caso que queremos evitar. Supongamos que desde una terminal T1 se selecciona la funcionalidad de eliminar plaza. Se busca una plaza libre a eliminar y se abre la ventana de confirmación: VConfirmarER (si es una plaza de aparcar) o VConfirmarER2 (si es una plaza de reserva). Simultáneamente se pretende, desde una segunda terminal T2 añadir un vehículo a esa plaza mediante la funcionalidad Reservar/Aparcar de la aplicación, de forma que se añade inmediatamente antes de que desde la terminal T1 se le dé al botón de eliminar en la ventana de confirmación (en una situación sin esta simultaneidad, la plaza a la que se le añade el vehículo no hubiese salido en la tabla de búsqueda para la eliminación). Ahora bien, para evitar que al pulsar el botón de eliminar desde la primera terminal se elimine la plaza (lo cual causaría un error de funcionamiento de la aplicación), debemos tratarlo con el nivel de aislamiento que merece esta situación. En este caso parece lógico establecer el modo *serializable* en la transacción de la eliminación.

Para tratar las transacciones con java hemos decidido crear los métodos siguientes:

- *public void commitTransaction()* : (en DAOUsuarios) envía la sentencia SQL "commit" de forma que permitirá cerrar una transacción que esté en curso desde un terminal en concreto.
- *public void beginTransaction()* : (en DAOUsuarios) envía la sentencia "begin transaction", que permite indicar el inicio de una transacción desde un terminal.
- *public void beginTransactionSerializable*: (en DAOUsuarios) envía la sentencia "begin transaction isolation level serializable" que permite iniciar una transacción con modo de aislamiento *serializable* desde un terminal.

Ahora toca decidir los puntos de inicio y fin de las transacciones para no cometer errores que

puedan entorpecer nuestra misión. Así pues, en cuanto a la **ventana de reservar y aparcar** se coloca el inicio de transacción en el botón de búsqueda (lo que quiere decir que al pulsar sobre este botón se llama al método de *beginTransaction()*). Para evitar errores en caso de que se pretenda usar de nuevo este botón, antes de llamar a este método se llama al método *commitTransaction()* al pulsar el botón. Por último cuando se pretende salir de esta ventana, se debe hacer mediante el botón de *volver*, ya que es este el que llama al método para hacer commit de la transacción. Ahora bien, en cuanto a la **ventana de eliminar plaza** (tanto la de aparcar como de reserva) el inicio de la transacción se sitúa en el botón de búsqueda, el cual al pulsar también finaliza la transacción anterior antes de iniciar la siguiente en curso de ese terminal (por si se pretende buscar varias veces), lo cual también permite que si ocurre un error en la anterior transacción podamos continuar en la ventana para volver a buscar. El fin de la transacción se encuentra, como en la ventana anterior, en el botón de volver, desde el cual se llama al método *commitTransaction().*

Una vez conocida la localización de las transacciones, se procede con el tratamiento del error *serializable* que surge de PostgreSQL (error que ya fue visto en clase de prácticas). El error es captado por los métodos *eliminarPlazaAparcar(int codigo)* y *eliminarPlazaReserva(int codigo)* de la clase 'DAOPlaza'. La idea es aprovecharse del tratamiento de los errores que nos ofrece Java. Es el botón de eliminar de las pestañas de confirmación (tanto de plazas aparcar como reservar) el que llama a este método a través primero de la fachada de la aplicación y esta a través de la fachada de la base de datos, que tiene como atributo un 'DAOPlaza'. De esta forma podemos lanzar la excepción desde el catch del método de 'DAOPlaza' (por lo tanto, en caso de producirse alguna excepción SQLException), para que lo capte el bloque try-catch que ponemos en el método (*eliminarPlazaAparcar* ó *eliminarPlazaReserva*) de la fachada de la base de datos, y lo vuelva a lanzar, y ser así recogido por el try-catch del método (*eliminarPlazaAparcar* ó *eliminarPlazaReserva*) de la fachada de la aplicación que, a su vez, de nuevo lo lanzará. Por último será el método de ActionPerformed de los botones eliminar de las ventanas de confirmación las que reciban el error. Entonces en caso de captar un error del tipo ExceptionSQL en este útimo método lo que se hace es valorar si es un error debido al modo serializable, lo que querría decir que entre que se ha pulsado el botón de búsqueda y el **último** botón de eliminar de la ventana de confirmación, otra transacción ha modificado la tupla que ahora queremos eliminar, y por lo tanto no la elimina.

Este error tiene el código '40001', por lo que podemos hacer la comparación *"40001".equals(error.getSQLState)* para comprobar si es el error buscado (recordemos que estamos en el método *eliminarActionPerformed* de la ventana VConfirmarER ó VConfirmarER). En caso afirmativo abrimos una ventana emergente que avisa sobre este error y se cierra la ventana de confirmación. Para evitar problemas también con eliminaciones que se puedan hacer de forma seguida, lo que se hace es, al pulsar el **primer** botón eliminar de las ventanas VEliminarReservar y VEliminarAparcar, reiniciar la tabla de búsquedas, para en caso de haber eliminado una plaza o añadido un vehículo a esa plaza, que no salga la lista desactualizada con esa plaza. Por ello al final del método *eliminarActionPerformed* de VEliminarReservar y VEliminarAparcar se inicia una nueva tabla de búsqueda.

Finalmente, situándonos en el caso anterior, si el terminal T1 ha seleccionado una plaza a

eliminar encontrándose ahora en la ventana de confirmación y la terminal T2 añade un vehículo a esa plaza, lo que sucede es que cuando T1 pulse el botón de eliminar, este botón se quede inhabilitado ("en espera") hasta que se confirma la transacción de T2 que sucede cuando se pulsa el botón de volver de la ventana de Reservar/Aparcar (de T2), y entonces se muestra la ventana emergente con el texto de aviso:

"ERROR debido al modo serializable:

Hay otra transacción en curso que ha modificado la misma plaza. Inténtelo de nuevo"

Y al cerrarlo se cierra la ventana de confirmación de la eliminación, apareciendo la ventana de eliminaciones con la tabla de búsqueda vacía. Ahora bien, si volvemos a buscar desde esta ventana plazas para eliminar (se hace commit de la transacción anterior y se inicia una nueva) ya no saldrá la plaza que queríamos eliminar porque ya está ocupada (desde la terminal T2 reservado o aparcado en esa plaza), y solo se pueden eliminar plazas vacías.

7. Funcionalidades y funciones

Funcionalidad 1 – Registrar que un coche ha sido aparcado o que una plaza ha sido reservada

Descripción general

En esta funcionalidad se va a llevar a cabo el registro de nuevas reservas de plazas, así como el estacionamiento de nuevos vehículos. Tiene como objetivo guardar en la base de datos la matrícula del vehículo, la plaza ocupada y la fecha de inicio. En el caso de reservar, se registrará también la fecha de fin.

Podemos dividir el desarrollo de la funcionalidad en dos fases consecutivas: búsqueda de plaza y registro de reserva/estacionamiento.

En la fase de búsqueda, se mostrarán todas plazas disponibles para reservar y para aparcar teniendo en cuenta que no se pueden reservar o aparcar plazas ya ocupadas, ni plazas ubicadas en el aparcamiento "ficticio" creado por defecto (para más información, ver Políticas de integridad referencial) Además, se permitirá filtrar por el id del Aparcamiento, por el código de Plaza o por el tipo de Vehículo.

Una vez seleccionada la plaza de reserva o de aparcar, comienza la fase de registro. Para cualquier tipo de plaza, se abrirá una nueva ventana que permita seleccionar el vehículo con el que se reserva/aparca. Se mostrará una tabla con todos los vehículos de usuarios no vetados (a excepción del vehículo "ficticio") y del tipo de la plaza seleccionada (C, M o G), que se podrá filtrar por matrícula. La fecha de inicio siempre será la fecha en la que se registre la reserva o el estacionamiento. Comprobará, además, que si el vehículo seleccionado es de un usuario no perteneciente a la USC, la diferencia entre ambas fechas no podrá ser mayor que siete días.

En segundo lugar, para aparcar, no se registrará la Fecha de Fin (se guardará como *NULL*). En este caso, no se mostrarán los vehículos ya aparcados.

Por último, cuando todos los datos necesarios hayan sido introducidos, se realizará el registro y se informará al usuario de la aplicación a través de un aviso de que se ha realizado con éxito la transacción.

Clases y métodos de Java y sentencias de SQL asociadas

A continuación, se exponen las diferentes clases relacionadas con la funcionalidad, organizadas por paquetes:

1. *aplication*
    - **FachadaAplicación**: permite comunicar a todos los paquetes. Se encuentran las funciones *obtenerVehiculosReserva, obtenerVehiculosNoAparcados, obtenerPlazasReserva, obtenerPlazasAparcar, hacerReserva, Aparcar* y *ObtenerRol*, los cuales llaman a sus correspondientes en la `**FachadaBaseDatos'.**
    - Las clases **RolUsuario, TipoUsuario**: son enumeraciones y se utilizan para saber los valores válidos que pueden tomar.

- Las clases **PlazaAparcar, PlazaReservar, Vehiculo**: se utilizan para construir las tablas.

2. *conexionBBDD*

- **FachadaBaseDatos**: responsable de la conexión con la base de datos, contiene todas las funciones mencionadas en **FachadaAplicación** y llama los DAOs donde están implementadas.
- **DAOPlazas**: contiene todas las funciones relacionadas con plazas que necesitan conectarse con la base de datos. Estas son:
  - *List<PlazaReserva> obtenerPlazasReserva(String codigoAparcamiento, Integer codigoPlaza, String tipoPlaza, boolean ocupadas):* devuelve una lista con todas las plazas de reserva (a excepción de la "ficticia"). Se pueden filtrar las plazas de forma que se correspondan con el código de Aparcamiento, el código de Plaza o el tipo de Plaza pasados como argumentos. El último argumento, si es false, provoca que solo se devuelvan las plazas disponibles.
  - *List<PlazaAparcar> obtenerPlazasAparcar(String codigoAparcamiento, Integer codigoPlaza, String tipoPlaza, boolean ocupadas):* análoga a la anterior, pero para plazas de tipo Aparcar.
  - *boolean hacerReserva(String matricula, int plaza, String aparcamiento, java.util.Date horaInicio, java.util.Date horaFin):* guarda en la base de datos una tupla en la relación Reservar con los datos pasados como argumentos. Devuelve true si se ha guardado correctamente en la base, false en caso contrario (lo que permite avisar al usuario del error).
  - *public boolean Aparcar(String matricula, int plaza, String aparcamiento):* análoga a la anterior pero guardando los datos en la relación de Aparcar. Se establece la fecha de inicio es la del momento en la que se introduzca en la base, la de fin se guarda como *NULL*

- **DAOUsuarios**: contiene todas las funciones relacionadas con usuarios o sus vehículos, que necesitan conectarse con la base de datos. Estas son:
  - *List<Vehiculo> obtenerVehiculos(String tipoletra):* devuelve una lista con todos los vehículos de usuarios no vetados (a excepción del vehículo ficticio). Solo obtiene los vehículos que sean del mismo tipo que tipoletra (C, M o G).
  - *List<Vehiculo> obtenerVehiculos(String dni, String tipoletra):* idéntica a la anterior pero permite filtrar por DNI del usuario del vehículo.
  - *public List<Vehiculo> obtenerVehiculosNoAparcados(String tipoletra):* devuelve una lista con todos los vehículos de usuarios no vetados(a excepción del vehículo ficticio) que no esten aparcados en el momento de realizar la consulta. Solo obtiene los vehículos que sean del mismo tipo que tipoletra (C, M o G).

- *public List<Vehiculo> obtenerVehiculosNoAparcados(String dni), String tipoletra:* idéntica a la anterior pero permite filtrar por DNI del usuario del vehículo.
- *String obtenerRol(String dni):* devuelve el rol del usuario que tenga el DNI pasado como parámetro.

3. *gui.* En este paquete se puede diferenciar entre tablas y ventanas JDialog:

- **VReservarAparcar:** JDialog que se puede acceder desde **VPrincipal**. El usuario puede introducir IdAparcamiento, codigoPlaza y tipoPlaza de la plaza que quiere reservar o registrar como ocupada, darle al botón '*Buscar*', seleccionar la plaza y acceder a otra ventana para reservar a través del botón '*Reservar*' o para aparcar, pulsando el botón '*Aparcar*'.
  - *Botón Buscar:* llama a *obtenerPlazasReserva* y *obtenerPlazasAparcar,* para actualizar las tablas que se muestran al usuario y que siguen el modelo **ModeloTablaPlazaReserva** o **ModeloTablaPlazaAparcar**, respectivamente. Filtra los resultados según IdAparcamiento, codigoPlaza y tipoPlaza, en caso de que hayan sido introducidos.
  - *Botón Reservar:* comprueba que se haya seleccionado una PlazaReserva y abre una nueva ventana **VHacerReserva.**
  - *Botón Aparcar:* comprueba que se haya seleccionado una PlazaAparcar y abre una nueva ventana **VAparcar.**
  - ***JTables jtableAparcar y jtableReservar:*** muestran los resultados de las funciones obtenerPlazasAparcar y obtenerPlazasReservar al dar al botón de Buscar.
- **VHacerReserva:** JDialog que se crea pasando como parámetros al constructor el IdAparcamiento, codigoPlaza y tipoPlaza seleccionado en la ventana **VReservarAparcar.** El usuario puede seleccionar el vehículo que va a ocupar la plaza desde una tabla y la fecha de fin de la reserva desde un jDateChooser(jcalendar). Se realiza la transacción al pulsar el botón *Reservar.*
  - *Botón Buscar:* llama a *obtenerVehiculos* para actualizar la tabla que se muestran al usuario y que siguen el modelo **ModeloTablaVehículo**. Filtra los resultados según el dni, en caso de que haya sido introducido en la caja de texto que muestra **VHacerReserva**.
  - *JDateChooser FechaFin:* muestra un calendario y permite seleccionar al usuario un día de fin. Comprueba que la fecha elegida no sea anterior a la fecha actual.
  - *Botón Reservar:* comprueba que se haya seleccionado un vehículo y una fecha de fin válida. Llama a *obtenerRol* para saber el rol del usuario del vehículo y, en caso de que no pertenezca a la USC, llama a una función privada *estanSeparadosPorMasDe7Dias* para comprobar que no reserve una plaza más de siete días (si es así, manda un aviso y no reserva). Por último, llama a la función *hacerReserva* para registrar la reserva en la base de datos. Muestra un

aviso al usuario sobre si se ha realizado con éxito la transacción o ha ocurrido algún error.

- *JTable jtablaVehiculos*: muestra los resultados de la función obtenerVehiculos al dar al botón de buscar.

- **VAparcar:** JDialog que se crea pasando como parámetros al constructor el IdAparcamiento, codigoPlaza y tipoPlaza seleccionado en la ventana **VReservarAparcar.** El usuario puede seleccionar el vehículo que va a ocupar la plaza desde una tabla. Se realiza la transacción al pulsar el botón *Aparcar.*
  - *Botón Buscar:* llama a *obtenerVehiculosNoAparcados* para actualizar la tabla que se muestran al usuario y que siguen el modelo **ModeloTablaVehículo**. Filtra los resultados según el dni, en caso de que haya sido introducido en la caja de texto que muestra **VAparcar**.
  - *Botón Aparcar:* comprueba que se haya seleccionado un vehículo. Llama a la función *Aparcar* para registrar el estacionamiento en la base de datos. Muestra un aviso al usuario sobre si se ha realizado con éxito la transacción o ha ocurrido algún error.
  - *JTable jtablaVehiculos*: muestra los resultados de la función obtenerVehiculosNoAparcados al dar al botón de buscar.
- **ModeloTablaPlazaReserva, ModeloTablaPlazaAparcar:** modelos para JTable, en los que se indican las columnas que se van a mostrar al usuario. Se utilizan en VReservarAparcar para mostrar las plazas.
- **ModeloTablaVehiculo:** modelo para JTable. Se utiliza en **VHacerReserva** y **VAparcar** para mostrar los vehículos.

Funcionalidad 2 – Historial de pagos

Descripción general

Esta funcionalidad consiste en un historial completo de todos los movimientos que han tenido lugar en el sistema de aparcamientos de la USC, tanto estacionamientos como reservas. La funcionalidad se muestra a través de una ventana que aparece al presionar el botón "Historial de pagos" en el menú principal de la aplicación. Esta ventana es, esencialmente, un buscador, que permite realizar búsquedas empleando diversos parámetros como la matrícula del vehículo, el DNI del propietario o el número de la plaza.

Además de recuperar los datos de los estacionamientos y las reservas, la funcionalidad calcula el tiempo total y el importe requerido por cada uno de estos. El importe, además, depende del rol del propietario del vehículo. Si este pertenece a la USC, se aplicará un descuento. Se puede dar el caso de que haya estacionamiento en la base de datos que no hayan finalizado aún, ya que el vehículo puede seguir aparcado. En este caso, la funcionalidad calcularía la duración del estacionamiento con la fecha y hora actuales. De la misma manera, el precio se mostraría actualizado con el tiempo total que el vehículo lleva estacionado hasta ese momento.

Cabe destacar que los campos de búsqueda requieren una sintaxis específica para recuperar los datos. Por ejemplo, si se quieren buscar los estacionamientos y reservas que han tenido lugar a partir de una fecha concreta, debe introducirse la fecha con el formato 'aaaa-mm-dd'. Para que no haya confusiones, algunos de los campos de búsqueda poseen un pequeño icono de una interrogación que explican la sintaxis necesaria.

Además de lo anterior, la funcionalidad también ofrece la posibilidad de retirar vehículos aparcados. Si en la tabla de Estacionamientos se selecciona una fila de un estacionamiento que no tenga fecha de salida (porque el vehículo aún está aparcado) y se presiona el botón 'Retirar Vehículo', se le asigna como fecha de salida el momento en el que se presiona el botón, y así el vehículo deja de estar estacionado.

Métodos de Java y sentencias de SQL asociadas

En lo referente a la implementación en Java, esta funcionalidad se encuentra en el JFrame "VPagos", que se encuentra en el paquete "gui". La ventana presenta cinco tipos de elementos principales:

- **Campos de búsqueda:** Son un total de diez JTextField. En ellos, se puede insertar texto para restringir los resultados de la búsqueda. Por ejemplo, el primero de ellos permite introducir una matrícula para obtener resultados de el vehículo con esa matrícula. Se permite buscar por matrícula del vehículo, DNI del usuario, ID del aparcamiento, número de la plaza, costes máximo y mínimo, fechas máxima y mínima y duraciones máxima y mínima del estacionamiento o reserva. Como ya se indicó, los campos de búsqueda requieren una sintaxis precisa para introducir los datos, que se muestra en iconos de interrogación al lado de sus respectivos campos de búsqueda.
- **Botón "Buscar":** Una vez introducidos todos los parámetros de búsqueda (que puede ser ninguno) en los campos de búsqueda, al presionar el botón "Buscar" se creará una consulta de SQL que contenga todas las restricciones especificadas. Esta consulta se ejecutará en la base de datos y sus resultados se mostrarán a modo de tuplas en dos JTable, uno para los estacionamientos y otro para las reservas.
- **JTables para estacionamientos y reservas:** Son dos JTable que muestran todos los resultados de la búsqueda. El JTable de estacionamientos emplea el modelo ModeloTablaAparcar. A su vez, el JTable de reservas hace uso del modelo ModeloTablaReservar.
- **Botón "Retirar Vehículo":** Si se selecciona una fila en la tabla de Estacionamientos que no tenga fecha de salida, se creará una instrucción se SQL para modificar este campo de la tupla en cuestión y darle el valor del momento actual (con now()). El propósito es retirar vehículos aparcados.
- **Botón "Volver":** Cierra la ventana y vuelve al menú principal.

A continuación se explicará el proceso que tiene lugar cuando se realiza una búsqueda con el botón "Buscar", así como cada método involucrado y su papel en la funcionalidad.

- *public void historialPagos():* ('FachadaAplicacion'). Es el método que inicia la funcionalidad. Se ejecuta al presionar el botón 'Historial de pagos' en la ventana principal de la aplicación. Llama al método 'historialPagos()' de 'FachadaGui' .

- *public void historialPagos():* ('FachadaGui'). Inicializa y hace visible la ventana 'VPagos' de la funcionalidad.
- *public void buscarPagos():* ('VPagos'). Es el método que se ejecuta cuando se presiona el botón 'Buscar' de 'VPagos'. Este método inicializa las dos JTable que van a mostrar los resultados de la consulta, creándolas con los modelos 'ModeloTablaAparcar' y 'ModeloTablaReservar'. Seguidamente, ejecuta el método 'setFilas()' sobre ambas tablas para pasarles los datos. Los datos los obtiene llamando a los métodos 'historialAparcar()' e 'historialReservar()' de la clase 'FachadaAplicacion'. Finalmente, si estos métodos devuelven listas vacías, se hace visible una etiqueta de aviso con el mensaje "No se encontraron resultados".
- *public List<Aparcar> historialAparcar(String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('FachadaAplicacion'). Este método es llamado por 'buscarPagos()' de la clase 'VPagos'. Recibe como argumentos todos los parámetros de búsqueda que se han introducido en los campos de búsqueda de 'VPagos'. Pueden ser todos vacíos, solo algunos, o ninguno. El método en si no realiza ninguna acción, su función pasar los parámetros de búsqueda hasta el método que realiza la consulta. Llama al método 'historialAparcar()' de la clase 'FachadaBaseDatos'. Recibe de este método una lista de objetos de la clase 'Aparcar' con los resultados de la consulta. Devuelve esta lista.
- *public List<Aparcar> historialAparcar (String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('FachadaBaseDatos'). Este método recibe los parámetros de búsqueda desde la fachada de la aplicación, y los pasa al método 'ConsultarHistorialAparcar()' de la clase 'DAOUsuarios', del cual recibe una lista de objetos de la clase 'Aparcar', la cual devuelve.
- *public List<Aparcar> ConsultarHistorialAparcar(String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('DAOUsuarios'). Es el método encargado de conectar con la base de datos para obtener los resultados deseados de la tabla 'Aparcar'. Recibe como argumentos los parámetros de búsqueda, esto es, las condiciones que deben cumplir todas las tuplas que se quieren obtener con la consulta. El método prepara una sentencia de tipo 'SELECT' para ejecutar en la base de datos, con todas las condiciones que se proveen. Si alguna de ellas está vacía, simplemente no la aplica, y obtiene tuplas con cualquier valor para ese campo. Hay dos datos, la duración y el coste, que no se almacenan en la base de datos, ya que son datos que pueden cambiar constantemente si el estacionamiento no ha finalizado todavía. Así, estos no se pueden aplicar a la consulta. En su lugar, se calculan en la clase 'Aparcar' y se comprueba que se cumplan las restricciones en este método. Al ejecutarse la consulta en la base de datos, se obtienen las tuplas, que se almacenan en instancias de la clase 'Aparcar'. Las que cumplan todas las restricciones se añaden a una lista, y esta lista es la que se devuelve.
- *public List<Reservar> historialReservar(String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('FachadaAplicacion'). Es análogo al método 'historialAparcar()' de la clase 'FachadaAplicacion', pero transfiere los parámetros de búsqueda a un método que

ejecuta una consulta sobre la tabla 'Reservar', en lugar de 'Aparcar'. En lugar de devolver una lista de objetos de la clase 'Aparcar', los devuelve de 'Reservar'.

- *public List<Reservar> historialReservar (String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('FachadaBasesDatos'). Este método es análogo al método 'historialAparcar()' de la clase 'FachadaBasesDatos'. En lugar de llamar a 'ConsultarHistorialAparcar()', llama a 'ConsultarHistorialReservar()'. En lugar de devolver una lista de objetos de la clase 'Aparcar', los devuelve de 'Reservar'.
- *public List<Reservar> ConsultarHistorialReservar(String dni, String mat, String pza, String ap, String cMax, String cMin, String dMax, String dMin, String fMax, String fMin):* ('DAOUsuarios'). Es la contrapartida de 'ConsultarHistorialAparcar()', pero ejecuta una consulta sobre la tabla 'Reservar' en lugar de 'Aparcar'. Recibe como argumentos los mismos parámetros de búsqueda que 'ConsultarHistorialAparcar()'. El método prepara una sentencia de tipo 'SELECT' con todas las condiciones provistas para ejecutar en la base de datos. De nuevo, si alguno de los parámetros está vacío, simplemente no se aplica la restricción a la consulta, y se obtienen tuplas con cualquier valor para ese campo. La duración y el coste se calculan en la clase 'Reservar', y se comprueba que se cumplan en este método. Los datos obtenidos como resultado de la consulta se almacenan en instancias de 'Reservar'. Las que cumplan todas las restricciones se añaden a una lista, y esta lista es la que se devuelve.

Tras la ejecución de estos métodos, el flujo se devuelve a 'buscarPagos()' de VPagos, el cual recibe las dos listas de instancias de 'Aparcar' y 'Reservar'. Proporciona estos datos a los JTable y finalmente se muestran por pantalla.

Veamos ahora el proceso que tiene lugar al pulsar el botón 'Retirar Vehículo'.

- *public void retirarVehiculo():* ('VPagos'). Este método se ejecuta al pulsar 'Retirar Vehículo'. Oculta todos los avisos (mensajes de error) y comprueba que haya una fila seleccionada, así como que la fila seleccionada no tenga nada en el campo 'Salida'. Si se cumplen estas condiciones, llama al método 'retirarVehiculo()' de 'FachadaAplicacion', pasándole como parámetros los valores de la matrícula, la plaza, el aparcamiento y la fecha de entrada de la fila seleccionada, puesto que estos son los atributos que conforman la clave primaria en la tabla Aparcar de la base de datos.
- *public void retirarVehiculo(String mat, String pza, String ap, String fi):* ('FachadaAplicacion'). Este método recibe los parámetros mencionados de la fila seleccionada y los pasa al método 'retirarVehiculo()' de 'FachadaBaseDatos'.
- *public void retirarVehiculo(String mat, String pza, String ap, String fi):* ('FachadaBasesDatos'). Este método, de nuevo, recibe los parámetros de la fila seleccionada y los pasa al método 'retirarVehiculo()' de 'DaoUsuarios'.
- *public void retirarVehiculo(String mat, String pza, String ap, String fi):* ('DaoUsuarios). Este método es el último de la cadena. Prepara una instrucción de SQL para actualizar la tupla de la tabla 'Aparcar' de la base de datos que tiene como clave primaria los valores recibidos. Cambia el valor del campo 'fechasalida' de null al momento actual, haciendo uso del método 'now()'. Así, se da a entender que el vehículo ya no está

aparcado, y queda guardada la tupla del estacionamiento para el registro histórico. Finalmente, se devuelve el flujo a 'VPagos'.

Al margen de estos dos procesos, hay un método más, que sirve para cerrar la ventana.

- *private void btnVolverActionPerformed(java.awt.event.ActionEvent evt):* ('VPagos'). Se ejecuta al presionar el botón 'Volver', y cierra la ventana.

Funcionalidad 3 - Registrar Infracciones

Descripción general:

Se trata de poder registrar una infracción cometida por un usuario en la base de datos. Incluye el registro del vetado en caso de llegar a las cinco infracciones, y, en cualquier caso (sea el usuario vetado o no), muestra el número de infracciones que ha cometido en total. Una vez que se veta a una persona, se reinician automáticamente el número de infracciones y se establece, por lo tanto, a cero.

El proceso se lleva a cabo en dos etapas: una de **búsqueda,** donde mediante la matrícula del vehículo se muestra el propietario en una tabla de búsqueda, y otra de **confirmación** en la cual el cliente puede confirmar el registro de la infracción.

Además, se crea un registro de actividad, que informa de las acciones que se están llevando a cabo así como de la actualización de los vetos que se describe a continuación.

Por último, la aplicación cada vez que se abre actualiza los vetados, esto es, comprueba para todos los usuarios que están vetados si han pasado más de 14 días desde la fecha de vetado hasta la apertura de la aplicación. En caso afirmativo, se elimina el veto. Los usuarios a los que se les ha eliminado el veto aparecerán reflejados en el registro de actividad en esta funcionalidad de *Registro de Infracciones* de forma automática.

Funciones de java relacionadas y sentencias SQL asociadas:

Respecto a la implementación en java, esta funcionalidad incluye al JFrame (implementamos la interfaz gráfica mediante Java Swing) llamado "VRegistroInfracciones". En ella, mediante distintos elementos (botones, listas...) se ejecutan ciertos métodos de java que permiten habilitar esta funcionalidad. Esto métodos se distribuyen a lo largo de los tres paquetes de la aplicación. Comenzaremos por los métodos principales del paquete 'conexionBBDD':

- *public void registrarInfraccion(String dni):* ('DAOUsuarios") registra una infracción en el usuario de dni [dni] estableciendo como fecha la fecha actual (mediante el Timestamp System.currentTimeMillis()). La petición enviada a la base de datos se realiza mediante un PreparedStatement, al cual se le asocia una consulta de actualización (un *update*). Como en el resto de métodos que se conectan con la base de datos se capturan los posibles errores que puedan surgir de esta conexión.
- *public void quitarVeto(String dni):* ('DAOUsuarios') quita un veto al usuario con dni [dni]. Esto es, establece a NULL el campo fechaveto de este mismo usuario en la tabla "Usuarios" de la base de datos. De nuevo se prepara una consulta con *update* para la misión de este método.

- *public void quitarTodasInfracciones(String dni)*: ('DAOUsuarios') pone a cero el número de infracciones del usuario cuyo dni es [dni]. Lo hace estableciendo el atributo numeroinfracciones a cero en la tabla "Usuarios" mediante una sentencia *update*.
- *public int mostrarNumeroInfracciones(String dni)*: ('DAOUsuarios') devuelve el número de infracciones cometidas por el usuario con dni [dni]. Se espera que el número se encuentre en el intervalo [0,5]. Además se devuelve –1 en caso de encontrar algún error a nivel de conexión con la base de datos. Se hace mediante una sentencia *select.*
- *public Usuario obtenerUsuario(String matricula)*: ('DAOUsuarios') devuelve un Usuario propietario del vehiculo cuya matrícula es la [matricula] que se pasa como argumento. Se busca en la base de datos dicho usuario (mediante una sentencia *select*), y se crea una instancia de la clase Usuario con los datos de la tupla devuelta., que será el objeto que se devolverá.
- *public ArrayList<Usuario> buscarUsuarioConVeto()*: ('DAOUsuarios') devuelve una lista con los usuarios que tienen actualmente un veto. Se trata de un método orientado a la actualización automática de los vetos. Mediante una consulta select se obtienen un conjunto de tuplas de la tabla usuario (los usuarios que no tienen NULL en el atributo fechaveto de la tabla) y para cada una de ellas se crea un Usuario. Se añaden todos a una lista, la cual será el objeto a devolver.

Ahora bien, todos estos métodos son llamados desde diversos métodos en la 'FachadaBaseDatos'. Algunos de estos últimos simplemente llaman a los métodos de 'DAOUsuarios': *public Usuario obtenerUsuario(String matricula)* y *public int mostrarNumeroInfracciones*. Otros métodos los emplean para cumplir su misión:

- *public int registrarInfraccion(String dni)*: ('FachadaBaseDatos') registra una infracción en el usuario con dni [dni] llamando al método *registrarInfraccion* del 'DAOUsuarios'. Además comprueba el número de infracciones (mediante *mostrarNumeroInfracciones* de 'DAOUsuarios'), y si es igual a 5, registra un veto (mediante *registrarVeto* de 'DAOUsuarios') y además establece a cero el número de infracciones del usuario con el método *quitarTodasInfracciones* del 'DAOUsuarios'.
- *public ArrayList<String> Actualizar()*: ('FachadaBaseDatos') devuelve la lista con los nombres de los usuarios cuyos vetos fueron quitados a causa de que pasaron más de 14 días desde la fecha del veto hasta el día en el que se abre la aplicación. Llama al método *buscarUsuariosConVeto()* y comprueba para cada usuario de la lista si la diferencia entre la fecha actual y la fecha del veto es mayor de 14 días. En caso afirmativo, se quita el veto (con *quitarVeto* de 'DAOUsuarios') y se pone el nombre en la lista que se va a devolver.

En el paquete 'aplication', esta funcionalidad solo se hace presente en la 'FachadaAplicacion', es esta la que permite comunicar a todos los paquetes. Simplemente se tienen los métodos *registrarInfraccion, mostrarNumeroInfracciones, obtenerUsuario* y *Actualizar*, los cuales llaman a sus correspondientes en la 'FachadaBaseDatos'. Esta clase implementa el main de la aplicación por lo que es en ella en la que se debe llamar al método *Actualizar()* de la instancia 'FachadaAplicacion' que se crea, justo después de la creación.

En el paquete 'gui', se implementan ciertas clases de relevancia para esta esta funcionalidad. Una de ellas es el 'ModeloListaString' que permite llevar a cabo es registro de actividad de la funcionalidad. Otra clase es el 'ModeloTablaUsuarios' que permite la creación de la tabla de búsqueda que también ofrece la funcionalidad. Resumimos a continuación los principales servicios del JFrame "VRegistroInfracciones":

- **Botón Buscar**: al presionar sobre el se elimina la tabla anterior de búsqueda, y a priori la deja vacía. A continuación llama al método *buscarUsuario* de esta misma clase, que recibe un parámetro [matricula] y lo que hace es crear una tabla de un solo elemento con el Usuario obtenido tras llamar all método *obtenerUsuario* de la instancia de 'FachadaAplicacion'. Si existe el usuario se muestra y en caso contrario se establece un aviso de que no se ha encontrado. Además si el usuario ya tiene registrado un veto se establece un aviso y además se inhabilita el botón siguiente.
- **Botón Confirmar Registro Infraccion:** al presionar sobre el (en caso de estar habilitado) se registra una infracción llamando método de *registrarInfraccion* de esta clase que llama al método *registrarInfraccion* de la clase 'FachadaAplicacion', y si este devuelve 5 (quiere decir que se ha establecido un veto), entonces se habilita el aviso del veto. Además se modifica el registro de actividad dependiendo de si se ha llegado al veto o no. Esto último es gestionado por el método *anhadirListaActividad,* que si recibe 1 como parámetro enonces se añade el texto "Registrado con éxito infracción en DNI [dni] que tiene un TOTAL de [numeroInfracciones] infracciones" en el registro de actividad; y si recibe 2 añade : "Alcanzado número máximo de infracciones, registrado veto en DNI [dni]". Por último, cabe mencionar que solo se puede añadir una infracción por búsqueda ya que este botón se inhabilita al ser pulsado, y se habilita solo en las búsquedas exitosas (si se encuentra el usuario y este no tiene veto).

Funcionalidad 4 – Introducir una plaza

Descripción general:

El objetivo de esta funcionalidad es el de añadir una plaza (ya sea de tipo aparcar o reservar) a un aparcamiento concreto ya existente en la base de datos. Se guardará en la base de datos el código de la plaza (que no puede coincidir con ninguno ya existente), el tipo de vehículo que va a utilizar esa plaza y el id del aparcamiento en el cual se quiere introducir.

Al presionar el botón "Configurar Plazas" el menú principal de la aplicación aparecerá otra ventana con la opción de "Añadir Plazas". A continuación, el usuario tendrá que escoger el tipo de plaza que quiere añadir a la base de datos (reservar/aparcar). Una vez lo haya seleccionado, se abre una última ventana donde se pide el código de la plaza, el tipo de vehículo y el id del aparcamiento. Al presionar el botón *añadir*, en caso de que la transacción se haya completado adecuadamente, se informará al usuario.

Cabe destacar que los administradores han de saber de antemano cuál es el id del aparcamiento en el que van a introducir la plaza, y qué código es válido de forma que no

coincida con alguna plaza existente. Aún así, en caso de error, el sistema notificará al usuario la imposibilidad de añadir dicha plaza debido a alguno de estos dos fallos.

Clases y métodos de Java y sentencias de SQL asociadas

La implementación en Java se realiza en tres paquetes diferentes:

1. *Aplication:* en este paquete se encuentran las clases en las que se basa la implementación. Destaca la 'Fachada Aplicación', que es la encargada de la comunicación con el resto de paquetes. Aquí aparecen los métodos *anadirPlaza* y *anadirPlazaReserva* que llamarán a su vez a los métodos correspondientes de 'FachadaBaseDatos'.

2. *ConexionBBDD:* en este paquete es donde se van a implememtar los métodos necesarios para realizar las consultas en la base de datos:

- **FachadaBaseDatos**: es el responsable de la conexión con la base de datos. Aquí aparecen de nuevo los métodos *anadirPlaza* y *anadirPlazaReserva* que llamarán a su vez a los métodos correspondientes de los DAOs donde están implementados .
- **DAOPlazas**: contiene todas las funciones relacionadas con las plazas que necesitan conectarse con la base de datos. Estas son:
  - *boolean anadirPlaza(String codigoAparcamiento, int codigoPlaza, String tipoPlaza):* guarda en la base de datos una tupla en la tabla PlazasAparcar con los datos pasados como argumentos. Es aquí donde se escribe la consulta SQL.

    ```
    BEGIN transaction;
    INSERT INTO PlazasAparcar (codigo, tipo, idAparcamiento) VALUES (?, ?, ?);
    COMMIT;
    ```

    Devuelve true si se ha guardado correctamente en la base, false en caso contrario(lo que permite avisar al usuario del error).
  - *public boolean anadirPlazaReserva(String codigoAparcamiento, int codigoPlaza, String tipoPlaza):* análoga a la anterior pero guardando los datos en la tabla de PlazasReserva.

    ```
    BEGIN transaction;
    INSERT INTO PlazasReserva (codigo, tipo, idAparcamiento) VALUES (?, ?, ?);
    COMMIT;
    ```

3. *Gui:* En este paquete aparecen dos ventanas JDialog:

- **VAnadir:** se puede accede desde **VPrincipal**. El usuario debe seleccionar el tipo de plaza que va a añadir, escogiendo el botón '*Aparcar*' o '*Reservar*' .
  - *Botón Aparcar:* abre una nueva ventana **VAnadirAparcar.**
  - *Botón Reservar:* abre una nueva ventana **VAnadirReservar.**

- **VAnadirAparcar.:** JDialog al que se accede pulsando '*Aparcar´* en la ventana **VAnadir.** El usuario debe rellenar el primer panel con el id del Aparcamiento, esoger el tipo de plaza en el menú desplegable, y finalmente rellenar el último panel con el código de la nueva plaza. Para añadir la plaza se deberá pulsar el botón *Anadir.*
    - *Botón Anadir:* comprueba que introducido un código de palza y aparcamiento válidos. Es decir, que el código sea un número entero y no esté repetido; y que el id del aparcamiento referencie a uno existente. Luego llama a *anadirPlaza* para añadir la plaza a la base de datos. Si la transacción se ha completado con éxito (o por el contrario ha ocurrido algún error) se informará al usuario.
- **VAnadirReservar:** Análogo al anterior pero para las plazas de reserva. JDialog al que se accede pulsando '*Reservar´* en la ventana **VAnadir.** El usuario debe rellenar el primer panel con el id del Aparcamiento, esoger el tipo de plaza en el menú desplegable, y finalmente rellenar el último panel con el código de la nueva plaza. Para añadir la plaza se deberá pulsar el botón *Reservar.*
    - *Botón Anadir:* comprueba que introducido un código de palza y aparcamiento válidos. Es decir, que el código sea un número entero y no esté repetido; y que el id del aparcamiento referencie a uno existente. Luego llama a *anadirPlazaReserva* para añadir la plaza a la base de datos. Si la transacción se ha completado con éxito (o por el contrario ha ocurrido algún error) se informará al usuario.
- **VAviso:** esta ventana aparece en caso de que se haya producido algun error.

Funcionalidad 5 – Eliminar una plaza

Descripción general:

El objetivo de esta funcionalidad es el de eliminar una plaza (ya sea de tipo aparcar o reservar) de la base de datos. En primer lugar habrá que buscar de plaza que se desea eliminar; y a continuación habrá que proceder con la eliminación de la misma.

Al presionar el botón "Configurar Plazas" el menú principal de la aplicación aparecerá otra ventana con la opción de "Eliminar Plazas". A continuación, el usuario tendrá que escoger el tipo de plaza que quiere eliminar de la base de datos (reservar/aparcar). Una vez lo haya seleccionado, comenzará la fase de busqueda.

Se abrirá otra ventana donde se mostrarán todas las plazas existentes (de reserva o de aparcamiento, en función de la opción seleccionada anteriormente) excepto las 'ficticias'. No obstante el sistema no mostrará aquellas plazas que estén siendo utilizadas en ese momento. Es decir, aquellas en las que haya algún vehículo estacionado o que tengan una reserva. Además, se permitirá filtrar por el id del Aparcamiento, por el código de Plaza o por el tipo de Vehículo para reducir las plazas que se muestran.

Una vez seleccionada la plaza de reserva o de aparcar, el usuario deberá pulsar el botón de *eliminar*. Se abrirá entonces una ventana que mostrará de nuevo la información de la plaza seleccionada; y preguntará al usuario si esa es la plaza que desea eliminar. Esta ventana de

confirmación resulta muy necesaria debido al carácter irreversible de la transacción. Es imprescindible que el usuario se asegure de que no está cometiendo un error.

Por último, cuando se procederá con la eliminación de la plaza y se informará al usuario si la transacción se ha realizado o no con éxito.

Clases y métodos de Java y sentencias de SQL asociadas

La implementación en Java se realiza en tres paquetes diferentes:

1. *Aplication:* en este paquete se encuentran las clases en las que se basa la implementación. Destaca la 'Fachada Aplicación', que es la encargada de la comunicación con el resto de paquetes. Aquí aparecen los métodos *eliminarPlazaAparcar* y *eliminarPlazaReserva* que llamarán a su vez a los métodos correspondientes de 'FachadaBaseDatos'.

2. *ConexionBBDD:* en este paquete es donde se van a implememtar los métodos necesarios para realizar las consultas en la base de datos:

- **FachadaBaseDatos**: es el responsable de la conexión con la base de datos. Aquí aparecen de nuevo los métodos *eliminarPlazaAparcar* y *eliminarPlazaReserva* que llamarán a su vez a los métodos correspondientes de los DAOs donde están implementados .
- **DAOPlazas**: contiene todas las funciones relacionadas con las plazas que necesitan conectarse con la base de datos. Estas son:
  - *List<PlazaReserva> obtenerPlazasReserva(String codigoAparcamiento, Integer codigoPlaza, String tipoPlaza, boolean ocupadas):* devuelve una lista con todas las plazas de reserva (a excepción de la "ficticia"). Se pueden filtrar las plazas de forma que se correspondan con el código de Aparcamiento, el código de Plaza o el tipo de Plaza pasados como argumentos. El último argumento, si es false, provoca que solo se devuelvan las plazas disponibles.
  - *List<PlazaAparcar> obtenerPlazasAparcar(String codigoAparcamiento, Integer codigoPlaza, String tipoPlaza, boolean ocupadas):* análoga a la anterior, pero para plazas de tipo Aparcar.
  - *boolean eliminarPlazaAparcar*(*int codigoPlaza):* elimina la base de datos una tupla de la tabla PlazasAparcar donde el código coincida con el del argumento. Es aquí donde se escribe la consulta SQL. Devuelve true si se ha eliminado correctamente de la base, false en caso contrario(lo que permite avisar al usuario del error).

```
BEGIN transaction;
DELETE FROM PlazasAparcar WHERE codigo = ?;
COMMIT;
```

- *public boolean eliminarPlazaReserva (int codigoPlaza):* análoga a la anterior pero eliminando la tupla de la tabla de PlazasReserva.

```
BEGIN transaction;
DELETE FROM PlazasReserva WHERE codigo = ?;
COMMIT;
```

3. *Gui:* En este paquete aparecen ventanas JDialog y una clase que permite la creación de la tabla de busqueda de plazas:

- **VEliminar:** se puede accede desde **VPrincipal**. El usuario debe seleccionar el tipo de plaza que va a eliminar, escogiendo el botón '*Aparcar*' o '*Reservar*'.
  - *Botón Aparcar:* abre una nueva ventana **VEliminarAparcar.**
  - *Botón Reservar:* abre una nueva ventana **VEliminarReservar.**
- **VEliminarAparcar.:** JDialog al que se accede pulsando '*Aparcar*' en la ventana **VEliminar.** El usuario puede introducir IdAparcamiento, codigoPlaza y tipoPlaza para filtrar la busqueda de la plaza que quiere eliminar y darle al botón '*Buscar*'. Luego deberá seleccionar la plaza y acceder a otra ventana confirmar la eliminación a través del botón '*Eliminar*'.
  - *Botón Buscar:* llama a *obtenerPlazasAparcar* para actualizar la tabla que se muestra al usuario y que sigue el modelo **ModeloTablaPlazaAparcar**. Filtra los resultados según IdAparcamiento, codigoPlaza y tipoPlaza en caso de que hayan sido introducidos.
  - *Botón Eliminar:* comprueba que se haya seleccionado una PlazaAparcar y abre una nueva ventana **VConfirmarER.**
- **VConfirmarER:** JDialog que se crea pasando como parámetros al constructor el IdAparcamiento, codigoPlaza y tipoPlaza seleccionado en la ventana **VEliminarAparcar.** El usuario puede ver los datos de la tabla que va a ser eliminada, y presionando el botón eliminar se llevará a cabo la transacción. De esta forma, si el usuario se ha equivocado al seleccionar la plaza (debido al caracter irreversible de la transacción) podrá pulsar el botón *volver* y seleccionar la plaza correcta.
  - *Botón Eliminar:* Llama a *eliminarPlazaAparcar* para eliminar la plaza de la base de datos. Si la transacción se ha completado con éxito (o por el contrario ha ocurrido algún error) se informará al usuario.
- **VEliminarReservar.:** Análogo al anterior pero para las plazas de reserva. JDialog al que se accede pulsando '*Reservar*' en la ventana **VEliminar.** El usuario puede introducir IdAparcamiento, codigoPlaza y tipoPlaza para filtrar la busqueda de la plaza que quiere eliminar y darle al botón '*Buscar*'. Luego deberá seleccionar la plaza y acceder a otra ventana confirmar la eliminación a través del botón '*Eliminar*'.
  - *Botón Buscar:* llama a *obtenerPlazasReserva* para actualizar la tabla que se muestra al usuario y que sigue el modelo **ModeloTablaPlazaReserva**. Filtra los resultados según IdAparcamiento, codigoPlaza y tipoPlaza en caso de que hayan sido introducidos.

- *Botón Eliminar:* comprueba que se haya seleccionado una PlazaReserva y abre una nueva ventana **VConfirmarER2.**

- **VConfirmarER2:** JDialog que se crea pasando como parámetros al constructor el IdAparcamiento, codigoPlaza y tipoPlaza seleccionado en la ventana **VEliminarReservar.** El usuario puede ver los datos de la tabla que va a ser eliminada, y presionando el botón eliminar se llevará a cabo la transacción. De esta forma, si el usuario se ha equivocado al seleccionar la plaza (debido al caracter irreversible de la transacción) podrá pulsar el botón *volver* y seleccionar la plaza correcta.
  - *Botón Eliminar:* Llama a *eliminarPlazaReserva* para eliminar la plaza de la base de datos. Si la transacción se ha completado con éxito (o por el contrario ha ocurrido algún error) se informará al usuario.
- **VAviso:** esta ventana aparece en caso de que se haya producido algun error.

- **ModeloTablaPlazaReserva, ModeloTablaPlazaAparcar:** modelos para JTable, en los que se indican las columnas que se van a mostrar al usuario. Se utilizan en VEliminarAparcar y en VEliminarReservar para mostrar las plazas.

Funcionalidad 6 - Añadir Usuario y Vehículo

Descripción general

Esta funcionalidad permite añadir nuevos usuarios a la base de datos, junto con sus respectivos vehículos. Al hacer clic en el menú en "Gestión usuarios" vemos la opción de añadir un usuario o registrar vehículos para un usuario ya existente. En el caso del nuevo usuario, se abre una pestaña donde se pide introducir toda la información del usuario (no se permiten datos nulos, ni por supuesto DNI repetido) y además existe otra pestaña donde se pueden in incluyendo vehículos del usuario en una lista. Cuando se esté satisfecho con la lista de vehículos, al hacer clic en "añadir" primero se introducirá el usuario y posteriormente, uno a uno, todos los vehículos. La pestaña de registrar vehículos es semejante solo que además se pide el DNI del usuario en cuestión, y trae incorporada una tabla con información de ese DNI para asegurar que es el correcto.

Métodos de Java y sentencias de SQL asociadas

Hemos dividido el código en 3 paquetes diferentes:

1. *Aplication.* Además de la fachada del paquete, que se encarga de llamar a la mayoría de las funciones, destacan aquí la propia implementación de las clases Usuario y Vehículo, que son las que vamos a rellenar de datos para posteriormente introducirlas en la BBDD (y además, las respectivas enumeraciones del Rol usuario y Tipo plaza vehículo). Por encima, se ha creado una nueva excepción ExcepcionAparcamiento a la que no se le ha dado demasiado uso, pero si se emplea en la guarda de comprobar que no se cambia nunca el dueño de un vehículo.
   Un método remarcable de la clase Usuario es incorporarVehiculos, que recibe una lista de vehículos de Java y los incorpora al arraylist del Usuario de uno en uno.
   Las únicas funciones de la fachada aplicación que nos incumben para esta funcionalidad

son registrarUsuario y registrarVehiculo, en las cuales entraremos en detalles posteriormente.

2. *ConexionBBDD*. Esta división del proyecto está a su vez dividida en la fachadaBD (que es llamada normalmente por la fachada de la aplicación) y varios DAOs (a los cuales llama la fachada). En cuanto a la incorporación de los usuarios, lo interesante va a ser la función registrarUsuario de la fachada, y la homónima dentro del DAO Usuarios. Aquí es donde siempre se realizan las consultas e inserciones de SQL: Igual que con el resto de funciones, se crea un *prepared statement* para poder enviárselo a la conexión de la base de datos. La característica del statement es que tiene "huecos" que podemos rellenar con datos, con esta estructura:

```
stmUsuario=con.prepareStatement("INSERT INTO Usuarios (dni, nombre, telefono, correo, fechaIngresoUSC, rol,
fechaVeto, numeroInfracciones) VALUES (?, ?, ?, ?, ?, ?, ?, ?)");
```

   De esta forma nos podemos evitar bastantes errores y manejar de forma mucho más sencilla la introducción de datos. La función añadir usuario, despues de introducir el Usuario a la base de datos, procede posteriormente a introducir uno a uno los Vehículos de este usando la misma técnica, pero recorriéndolos iterativamente uno a uno.

   Existe entonces, además, esta misma funcionalidad pero con la introducción de vehículos, y de hecho registrarUsuario llama en cada vehículo a registrarVehículo.

3. *Gui*. En la interfaz gráfica no solo se introducen los datos sino que además se manejan y se meten en un objeto Usuario, el cual se acaba enviando al DAO para introducirse en la base de datos:
   - **VNuevoVehiculo** y **VNuevoUsuario**: Estas dos pestañas son bastante parecidas en el sentido de que ambas crean un objeto a base de leer los datos que se le hayan introducido. La primera es la que se llama directamente de VGestionUsuarios, teniendo campos para toda la información del usuario (por ejemplo, una choose box para el tipo, o un JCalendar para la fecha de ingreso). Se ve además una lista con los vehículos del usuario, cuyo modelo se describe a continuación. Además de leer todos los datos de los JTextFields, la lista de vehículos se le debe pasar al Usuario con el método que ya habíamos mencionado antes incorporarVehiculos. Al final, se llama a la fachada aplicacion que lleva el control al DAO Usuarios para realizar la funcionalidad que se ha mencionado en el punto 2.
   - **VGestionUsuarios**: Ventana simple en la que figuran las dos opciones que llaman a VNuevoVehiculo y VAnadirUsuario, respectivamente
   - **ModeloListaVehiculos**: Este modelo para la JList permite guardar los vehículos que se han ido añadiendo y mostrarlos por pantalla. Se utiliza en VNuevoUsuario para incorporar al usuario todos los vehículos que se añaden uno a uno en VNuevoVehiculo
   - **VAnadirUsuario**: Es una pestaña que extiende a VNUevoVehiculo. En la anterior, no era necesario un campo DNI usuario ya que era un Dialog llamado por VNuevoUsuario. Ahora se le ha incorporado ese campo y además un botón de buscar que muestra información de usuario cuyo DNI es el introducido y así poder hacer la

comprobación con anterioridad.

Funcionalidad 7 - Estadísticas

Descripción general

La funcionalidad de estadística reporta información relevante sobre el uso del sistema de aparcamiento por parte de los diversos usuarios. Esta funcionalidad utiliza una serie de consultas SQL para extraer la información de interés de la base de datos y estos son mostrados en su respectiva ventana. Nos interesa recoger tres datos estadísticos de intereses diferentes:

- La primera, muestra el tiempo medio que los usuarios del aparcamiento dejan su vehículo estacionado, sean miembros de la USC o no.
- La segunda, imprime el número de veces que cada usuario ha aparcado y ha reservado plaza en el aparcamiento. Los resultados se ordenarán de manera descendente, priorizando la cantidad de veces que se ha aparcado en el aparcamiento
- La tercera, devuelve cuatro tablas diferentes acerca del uso de las plazas. Estas tablas reportan información de las plazas más y menos aparcadas y más y menos reservadas.

Los datos de los tres casos se operan de manera similar: se ejecuta una consulta SQL a la base de datos, los datos obtenidos por la consulta se limpian, si es que sea necesario, y por último se empaquetan los datos y se muestran a través de la interfaz gráfica a su debida manera.

Funciones de java relacionadas y sentencias SQL asociadas:

Una vez que se crea el JFrame correspondiente a esta funcionalidad, se procede a calcular todas las estadísticas necesarias. En el JFrame "VEstadisticas" se alojan dos funciones relevantes a esta funcionalidad:

- *private void calcStats():* este método se llama dentro del constructor de "VEstadisticas". La función se dedica a recoger los datos que le reportan diversas llamadas a funciones de un DAO y a mostrarlas en los diferentes elementos de la interfaz gráfica.
- *private void actTablaPlazas():* este método actualiza la tabla del último caso de interés estadístico según el usuario decida ver las plazas más/menos aparcadas/reservadas.

Las funciones que están en su respectivo DAO son las encargadas de obtener los diversos datos de la base de datos, todos los casos estadísticos comparten la manera de hacer consultas a la base de datos. Primeramente se establece una conexión a la base de datos, a partir de esta conexión se crea un objeto Statement y sobre este se ejecuta una consulta SQL mediante el método executeQuery(String query), la consulta devuelve un objeto ResultSet sobre el cual se itera hasta que todos los datos hayan sido preparados para devolverlos a la interfaz gráfica. A continuación, se muestran las diversas funciones que se encargan de obtener los datos y su respectiva devolución.

- *public String[] statsTmedioAparcar():* esta función se encarga de realizar el primer caso de interés estadístico. La consulta devuelve dos filas, si hay datos para ambos miembros

y no miembros de la USC, si no existen datos para alguno de los dos tipos de roles no existirá fila en la consulta. Sobre el ResultSet obtenido por la consulta, se obtiene, por cada fila, el valor de la columna 'rol', es un int que adopta 0 si es el resultado de los miembros de la USC o 1 si es el resultado de los usuarios que no son miembros de la USC, el valor de la columna 'tiempomedioaparcado' informa del tiempo medio del tipo de usuario, este valor es un String y debe someterse, si procede, a una etapa de limpieza. Cada fila se almacena en un String[2] donde el índice del dato de valor medio viene dado por el valor de la columna 'rol'.

- *public List<HashMap<String, Object>> statsVecesUsuario():* esta función se encarga de realizar el segundo caso de interés estadístico. La consulta reporta tantas filas como usuarios hayan aparcado/reservado y cuatro columnas con respectivos datos. Sobre el ResultSet obtenido por la consulta, se obtiene, por cada fila, el valor de la columna 'nombre', la columna 'dni', la columna 'vecesaparcado' y la columna 'vecesreservado'. Cada fila conforma un HashMap, todos estos se añaden a una lista de retorno que se devuelve.
- *public HashMap<String, List<HashMap<String,Object>>> statsPlazas()):* esta función se encarga de realizar el tercer y último caso de interés estadístico. La consulta agrupa plazas en cuatro categorías: más aparcadas, menos aparcadas, más reservadas y menos reservadas, aparte de la categoría cada fila tiene tres columnas más de datos. Sobre el ResultSet obtenido de la consulta, se obtiene, por cada fila, el valor de la columna 'plaza', la columna 'tipo', la columna 'veces' y la columna 'categoria'. Por cada categoría diferente se crea una clave en el HashMap principal que se va a devolver, cada valor de este HashMap es la lista de todas las plazas que pertenecen a esa categoría.

Para poder mostrar las tablas correspondientes a los dos últimos casos estadísticos se incluyen en el paquete 'gui' las respectivas clases 'ModeloTablaEstadisticaUsuarios' y 'ModeloTablaEstadisticaPlazas' que manejan la creación, relleno y gestión de las tablas.

Funcionalidad 8 - Autentificación

Descripción general

Esta funcionalidad consiste en una ventana destinada a ser la vía de entrada a la aplicación. Se trata de una ventana que pide al usuario de la aplicación un dni y una contraseña. Como la aplicación está destinada a ser usada por administradores de la USC, entonces necesariamente para poder acceder a la aplicación se exige que el dni sea de un usuario cuyo rol sea 'Administracion', y la contraseña que se introducide debe corresponder con la contraseña del usuario, almacenada en SHA256 en la base de datos. Si la autentificación es exitosa, se cierra la ventana de autentificación y se abre la ventana principal de la aplicación.

Funciones de java relacionadas y sentencias SQL asociadas:

La ventana asociada a esta funcionalidad es el JDialog VAutentificacion, que es instanciada desde la FachadaGui. Cuando se inicia la aplicación y se llama al método *iniciaVista()* de esta última clase, se crea un objeto de la ventana VAutentificacion. En esta ventana el elemento más destacable es el botón **Aceptar**, que al ser pulsado llama al método *comprobarAutentificacion* pasando como argumento el texto escrito por el usuario en el JTextField:

- *public Usuario validarUsuario(String dni)*: (en DAOUsuarios) obtiene los datos de un

usuario en la base de datos, a traves de su dni [dni]. La consulta que se realiza a la base de datos es un "select" que busca al usuario por el dni. Seguidamente se crea la instancia del Usuario, que es la que se va a devolver.

Para que VAutentificacion pueda acceder a este método, se crean los métodos correspondientes en las clases FachadaAplicacion (*comprobarAutentificacion(String dni)*) y FachadaBasesDatos (*validarUsuario(String dni)*), que simplemente devuelven el Usuario devuelto del método descrito. Si el usuario devuelto no es nulo, el rol es de 'Administracion' (*u.getRol()==RolUsuario.Administracion*) y la contraseña introducida por el usuario, después de realizar su respectivo hash SHA256, corresponde con la contraseña del usuario almacenada en la base de datos, entonces cierra la ventana de la autentificación, se establece, mediante el setter el atributo Usuario de la fachada de la aplicación y finalmente se actualiza la etiqueta de bienvenida de VPrincipal (parent de VAutentificacion) del usuario con su nombre. En caso de no haber encontrado el usuario no se permite avanzar y simplemente se muestra un mensaje de error.

[ANEXO] Script de generación de la Base de Datos

```
CREATE TABLE Usuarios (
dni CHAR( 9 ) ,
nombre VARCHAR( 500 ) NOT NULL,
telefono CHAR( 9 ) ,
correo VARCHAR ( 500 ) ,
fechaIngresoUSC DATE,
rol VARCHAR( 50 ) NOT NULL,
fechaVeto DATE,
numeroInfracciones INT NOT NULL DEFAULT 0 ,
contrasena VARCHAR(64) default NULL,
  CHECK ( numeroInfracciones IN ( 0 , 1 , 2 , 3 , 4 , 5 ) ) ,
  CHECK ( rol IN ( 'Administracion' , 'Docente' , 'Alumno', 'noUSC' ) ) ,
  PRIMARY KEY ( dni )
);

CREATE TABLE Vehiculos (
matricula CHAR( 7 ) ,
tipo CHAR( 1 ) NOT NULL,
marca VARCHAR( 500 ) ,
modelo VARCHAR( 500 ) ,
anhoMatriculacion NUMERIC( 4 , 0 ) ,
dni CHAR( 9 ) DEFAULT '000000000' ,
PRIMARY KEY ( matricula ) ,
```

```
FOREIGN KEY ( dni ) references usuarios(dni)
ON DELETE SET NULL
ON UPDATE CASCADE,
CHECK ( anhoMatriculacion > 1900 ) , /* Primer coche matriculado en Espana */
CHECK ( tipo IN ( 'C' , 'M' , 'G' ) ) /* Coche , moto o vehiculo grande */
);

--Vehiculo ficticio para el setDefault
INSERT INTO vehiculos (matricula, tipo, marca, modelo, anhoMatriculacion, dni)
VALUES ('0000AAA', 'C', null, null, null, null);
CREATE TABLE Aparcamientos (
id CHAR(5),
direccion VARCHAR( 1000 ) ,
aforo numeric(3,0) NOT NULL,
PRIMARY KEY ( id )
);

--Aparcamiento ficticio para el setDefault
INSERT INTO Aparcamientos (id, direccion, aforo)
VALUES ('PARK0', null, 2);

CREATE TABLE PlazasAparcar (
codigo NUMERIC( 3 , 0 ) ,
tipo CHAR( 1 ) NOT NULL,
idAparcamiento CHAR(5),
PRIMARY KEY ( codigo, idAparcamiento ) ,
FOREIGN KEY ( idAparcamiento ) references aparcamientos(id)
ON DELETE NO ACTION
ON UPDATE CASCADE,
CHECK ( tipo IN ( 'C ' , 'M' , 'G' ) ) /* Coche , moto o vehiculo grande */
);

--Plaza ficticia para el SetDefault
INSERT INTO PlazasAparcar (codigo, tipo, idAparcamiento)
VALUES (0,'C','PARK0');

CREATE TABLE PlazasReserva (
codigo NUMERIC( 3 , 0 ) ,
```

```
tipo CHAR( 1 ) NOT NULL,
idAparcamiento CHAR(5),
PRIMARY KEY ( codigo, idAparcamiento ) ,
FOREIGN KEY ( idAparcamiento ) references aparcamientos(id)
ON DELETE NO ACTION
ON UPDATE CASCADE,
CHECK ( tipo IN ( 'C' , 'M' , 'G' ) ) /* Coche , moto o vehiculo grande */
);

--Plaza ficticia para el SetDefault
INSERT INTO PlazasReserva (codigo, tipo, idAparcamiento)
VALUES(0,'C','PARK0');


CREATE TABLE aparcar (
matriculaVehiculo CHAR( 7 ) ,
codigoPlaza NUMERIC( 3 , 0 ) DEFAULT 0 ,
idAparcamiento CHAR(5),
fechaentrada TIMESTAMP,
fechasalida TIMESTAMP,
PRIMARY KEY ( matriculaVehiculo , codigoPlaza , idAparcamiento, fechaentrada ) ,
FOREIGN KEY ( matriculaVehiculo) references vehiculos(matricula)
ON DELETE SET DEFAULT
ON UPDATE CASCADE,
FOREIGN KEY ( codigoPlaza, idAparcamiento ) references plazasAparcar(codigo, idAparcamiento)
ON DELETE SET DEFAULT
ON UPDATE CASCADE
);

ALTER TABLE aparcar
ALTER COLUMN matriculaVehiculo SET DEFAULT '0000AAA';

ALTER TABLE aparcar
ALTER COLUMN codigoPlaza SET DEFAULT 0;

ALTER TABLE aparcar
ALTER COLUMN idAparcamiento SET DEFAULT 'PARK0';
```

```
CREATE TABLE reservar (
matriculaVehiculo CHAR( 7 ) ,
codigoPlazaReserva NUMERIC( 3 , 0 ) DEFAULT 0 ,
idAparcamiento CHAR(5),
fechaInicio TIMESTAMP,
fechaFin TIMESTAMP,
PRIMARY KEY ( matriculaVehiculo , codigoPlazaReserva , idAparcamiento, fechaInicio) ,
FOREIGN KEY ( matriculaVehiculo) references vehiculos(matricula)
ON DELETE SET DEFAULT
ON UPDATE CASCADE,
FOREIGN KEY ( codigoPlazaReserva, idAparcamiento ) references plazasReserva(codigo, idAparcamiento)
ON DELETE SET DEFAULT
ON UPDATE CASCADE,
CHECK( fechaInicio < fechaFin )
);

ALTER TABLE reservar
ALTER COLUMN matriculaVehiculo SET DEFAULT '0000AAA';

ALTER TABLE reservar
ALTER COLUMN codigoPlazaReserva SET DEFAULT 0;

ALTER TABLE reservar
ALTER COLUMN idAparcamiento SET DEFAULT 'PARK0';
```