Capítulo **3**

MODELO DE DISEÑO

3.1 Arquitectura Modelo-Vista-Presentador

La arquitectura *Modelo-Vista-Presentador* (MVP) [11] separa el modelo, la presentación y las acciones basadas en la interacción con el usuario en tres clases separadas. La vista le delega a su presentador toda la responsabilidad del manejo de los eventos del usuario. El presentador se encarga de actualizar el modelo cuando surge un evento en la vista, pero también es responsable de actualizar a la vista cuando el modelo le indica que ha cambiado. Por su parte, el modelo no conoce la existencia del presentador, por lo tanto, si el modelo cambia por acción de algún otro componente que no sea el presentador, debe "disparar" un evento para que el presentador se entere. Como se puede apreciar en la figura 3.1, a la hora de implementar esta arquitectura, se identifican los siguientes componentes:

- **IVista:** es la interfaz con la que el *Presentador* se comunica con la vista.
- **Vista:** vista que implementa la interfaz *IVista* y se encarga de manejar los aspectos visuales. Mantiene una referencia a su *Presentador*, al cual le delega la responsabilidad del manejo de los eventos.
- **Presentador:** contiene la lógica para responder a los eventos y manipula el estado de la vista mediante una referencia a la interfaz *IVista*. El Presentador utiliza el modelo para saber cómo responder a los eventos y es responsable de establecer y administrar el estado de una vista.
- **Modelo:** está compuesto por los objetos que conocen y manejan los datos dentro de la aplicación.

3.2 Diagramas de secuencia

A la hora de establecer qué métodos se necesitan en el modelo y en el presentador, se debe partir del prototipo de la interfaz de usuario presentado en el capítulo *Modelo de requisitos*. Analizando los posibles usos que tiene la aplicación, se obtienen

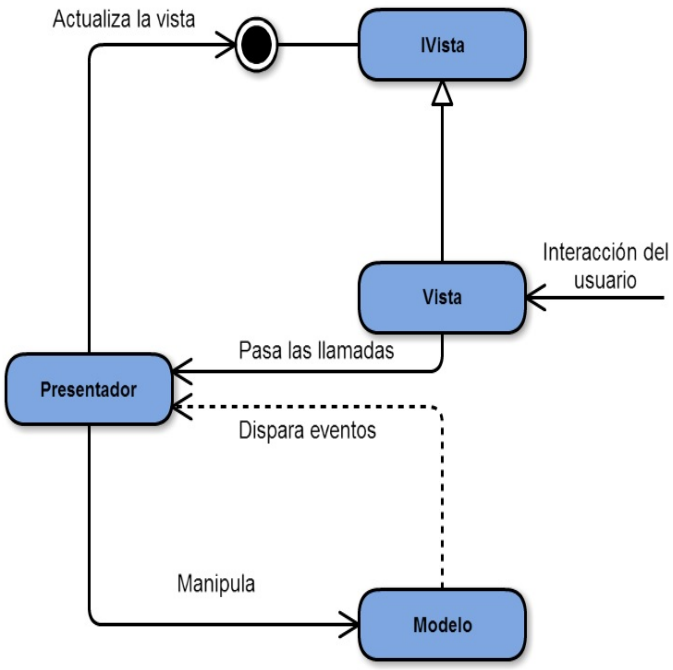

Figura 3.1: Diagrama de la arquitectura*Modelo-Vista-Presentador*

unos diagramas de secuencia determinados. Los diagramas de secuencia describen el uso de la aplicación según los eventos enviados entre los objetos de la arquitectura *MVP*. El diagrama de secuencia describe aspectos dinámicos de un sistema, a diferencia de los diagramas de clases que muestran información estática. Por esta razón, los diagramas de secuencias utilizan objetos mientras que los diagramas de clases utilizan clases como elementos básicos.

Cada objeto en el diagrama se representa con una línea vertical, que corresponde al eje temporal, donde el tiempo avanza hacia abajo. En este diagrama se muestran los eventos que ocurren en el tiempo, los cuales son enviados de un objeto a otro. El orden de los objetos no es importante. Lo importante es el orden en el que ocurren los eventos y la dependencia entre ellos, es decir, qué consecuencias tiene el envío de un evento.

Los mensajes enviados entre objetos corresponden con los métodos que hay que definir en las interfaces implementadas por la clases de esos objetos. Así, un mensaje enviado entre un objeto vista y un objeto presentador se representa mediante una flecha que va desde la línea vertical del objeto vista, hacia la línea vertical del objeto presentador, y tiene un identificador que corresponde con el nombre del método. En el diagrama de secuencia no se muestran los datos enviados o recibidos, sino que sólo se muestran los identificadores de los mensajes enviados. En los siguientes apartados se analizan detalladamente cada uno de estos mensajes y se indican los parámetros y el tipo de datos que devuelven (si es el caso).

En este apartado se presentan los diagramas de secuencia más significativos de la *Aplicación Android para pedir cita previa en peluquerías*. Así, en la figura 3.2 se muestra el flujo de mensajes que se producen entre la vista, el presentador y el modelo cuando el usuario quiere obtener información de una peluquería. En la secuencia se ha supuesto el caso de uso *Mostrar Peluquerías*.

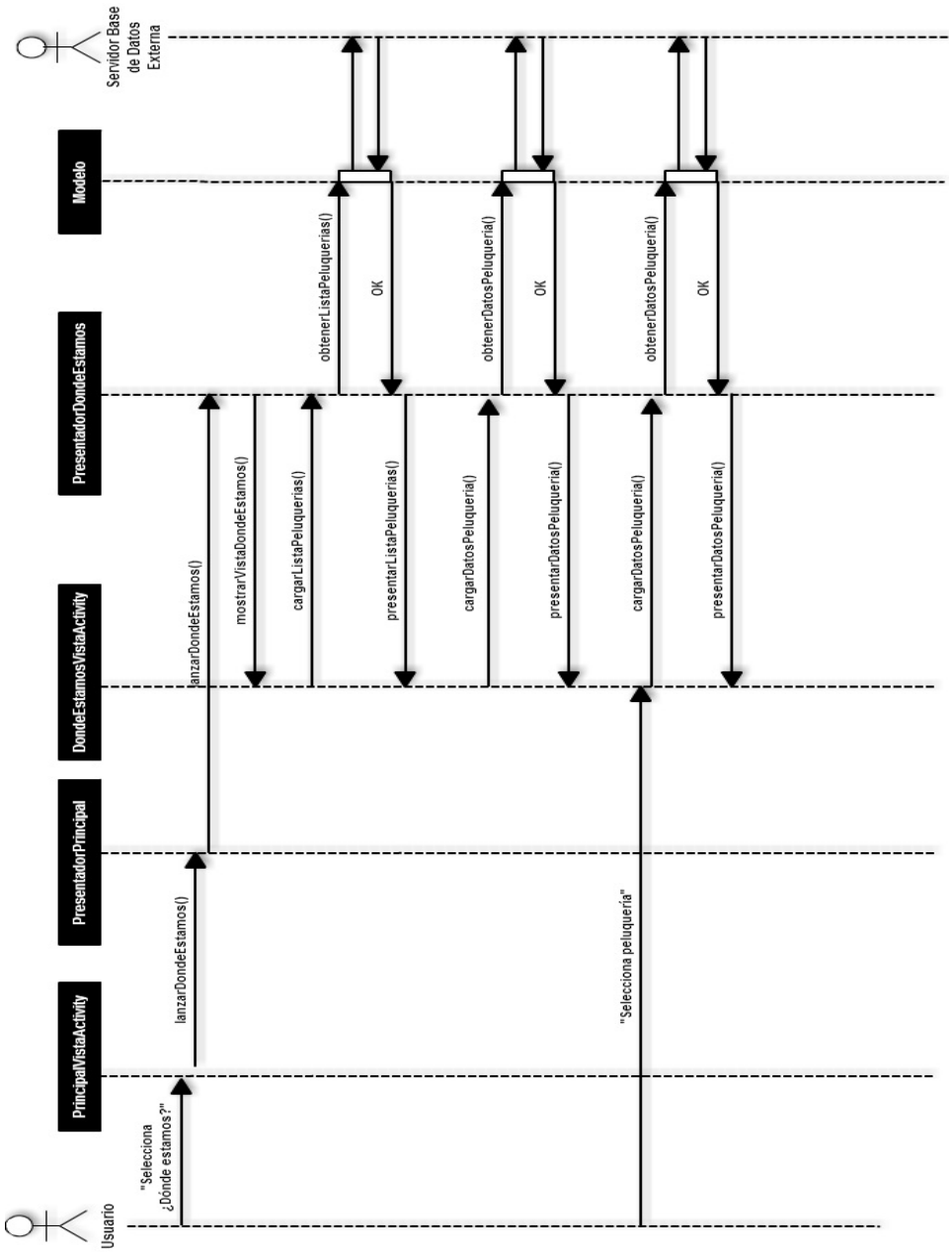

Figura 3.2: Diagrama de secuencia del caso de uso *Mostrar Peluquerías*

En la figura 3.3 se muestra el flujo de mensajes que se producen entre las vistas, presentadores y el modelo cuando el usuario, una vez elegida la peluquería, desea ver un mapa con la posición de dicha peluquería.

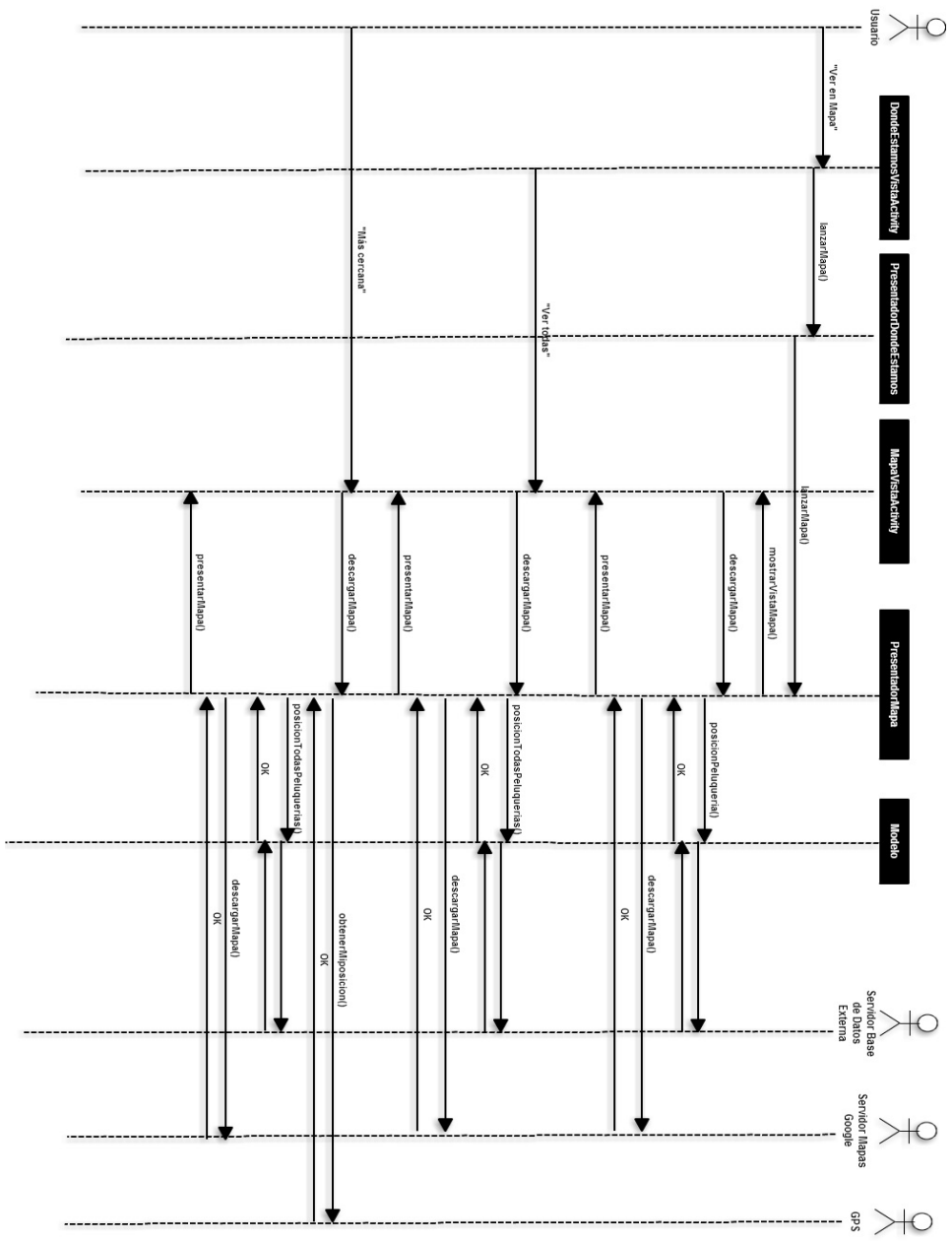

Figura 3.3: Diagrama de secuencia del caso de uso *Mostrar Mapa*

En la figura 3.4 se muestra el flujo de mensajes que se producen entre las vistas, presentadores y el modelo cuando el usuario intenta pedir una cita a través de la

aplicación. Éste es un diagrama de secuencia muy importante puesto que la finalidad principal de la aplicación es pedir cita.

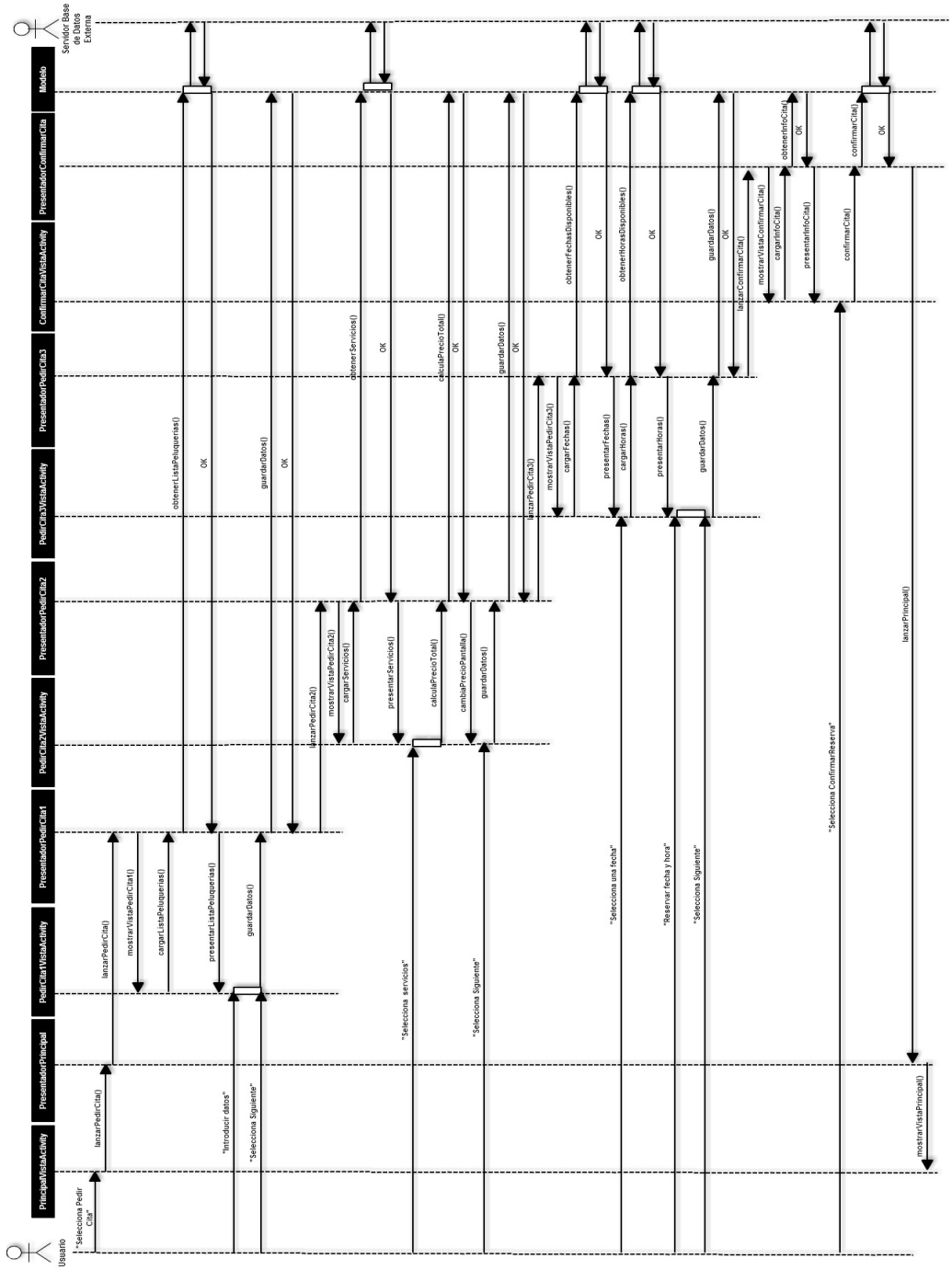

Figura 3.4: Diagrama de secuencia del caso de uso *Pedir Cita*

Por último, en la figura 3.5 se ve el flujo de mensajes del caso de uso *Mostrar Citas* que es el encargado de mostrar las citas que ha pedido el usuario mediante la aplicación.

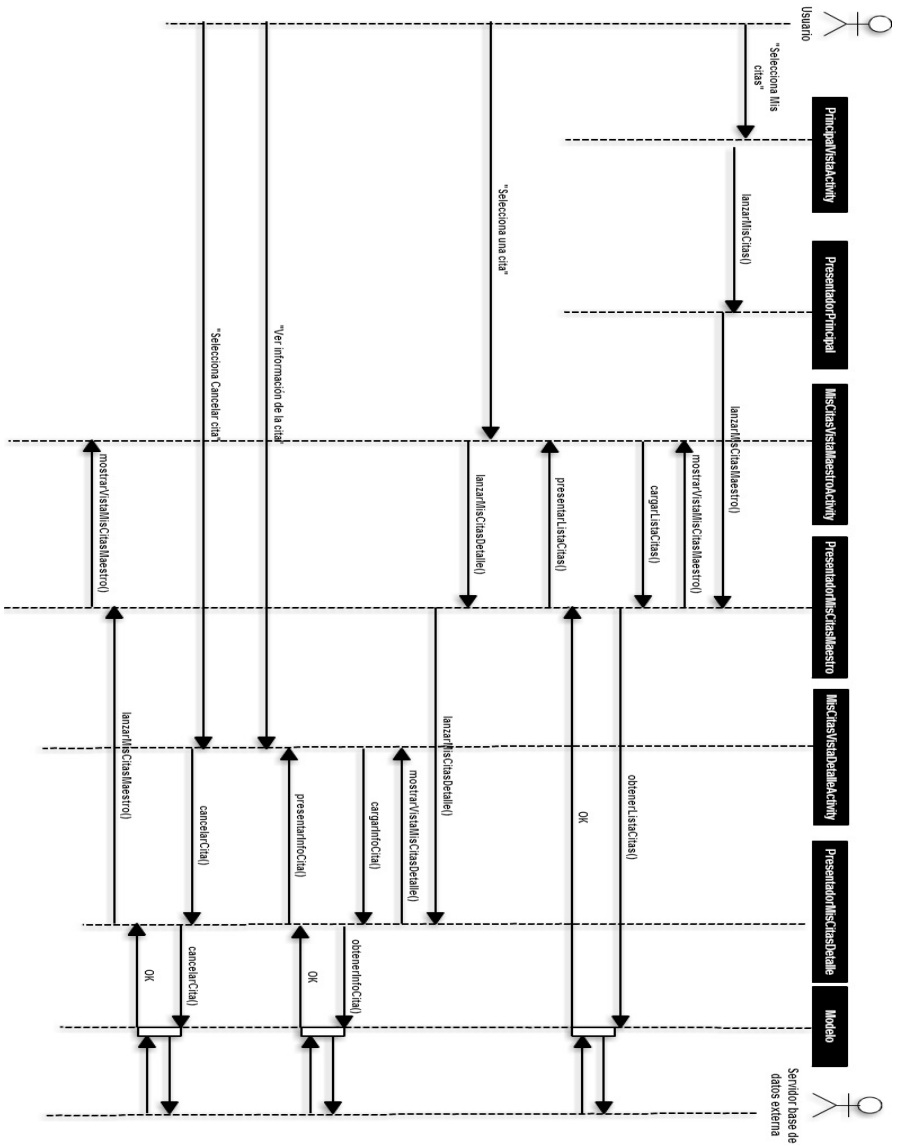

Figura 3.5: Diagrama de secuencia del caso de uso *Mostrar Citas*

3.3 Modelo de datos

Un modelo de datos es la descripción de la información que se utiliza en una aplicación. En el caso de la *Aplicación Android para pedir cita previa en peluquerías*,

la información se almacenará en la nube (base de datos externa) ya que ésta debe estar disponible para todos los usuarios de la aplicación. Para que la información esté disponible en cualquier momento y en cualquier lugar, debe estar almacenada en un servidor diseñado y mantenido por el desarrollador o por una entidad externa (opción escogida). Las bases de datos en la nube pueden estar basadas en *SQL* o utilizar un modelo de datos *NoSQL*. Concretamente, en este trabajo se utilizará una base de datos en la nube *NoSQL* que se caracteriza, entre otras cosas, por la ausencia de esquema, es decir, no se diseñan las tablas ni la estructura de los datos por adelantado [12]. Sin embargo, y como se conoce el tipo de información a almacenar, se puede realizar una descripción de la estructura de los datos y sus tipos. Así, para el diseño de la base de datos que utilizará la *Aplicación Android para pedir cita previa en peluquerías*, se presentan a continuación una serie de tablas en las que se definen:

- **Campo:** en donde se definirá su nombre.
- **Clave:** definirá si el campo es clave primaria o foránea (en este caso se indicará entre corchetes a qué otra tabla hace referencia en la forma *ClaveForánea [TablaExterna]*). Aunque el concepto de clave no tiene significado en una base de datos *NoSQL*, si es importante de cara a la aplicación desarrollada.
- **Tipo:** definirá el tipo del campo, y si acepta o no, que el campo esté vacío. Por defecto no existirán campos nulos, a excepción que se indique en las tablas (con la palabra *null*).

3.3.1 Diseño de la base de datos

La base de datos estará compuesta por un total de seis tablas: **Peluquerias, Horarios, Festivos, Citas, Servicios** y **CitaServicio**. A continuación se procede a definir cada tabla de la base de datos y sus campos como se puede ver en la figura 3.6.

3.3.1.1. Entidad Peluquerias

Entidad que contiene las peluquerías que permitan pedir cita a través de la *Aplicación Android para pedir cita previa en peluquerías*. Esta entidad cuenta con seis campos:

- **id_peluqueria:** es un campo de tipo *String* y clave primaria. Almacena un identificador único de la peluquería.
- **direccion:** es un campo de tipo *String* y almacena la dirección de la peluquería.

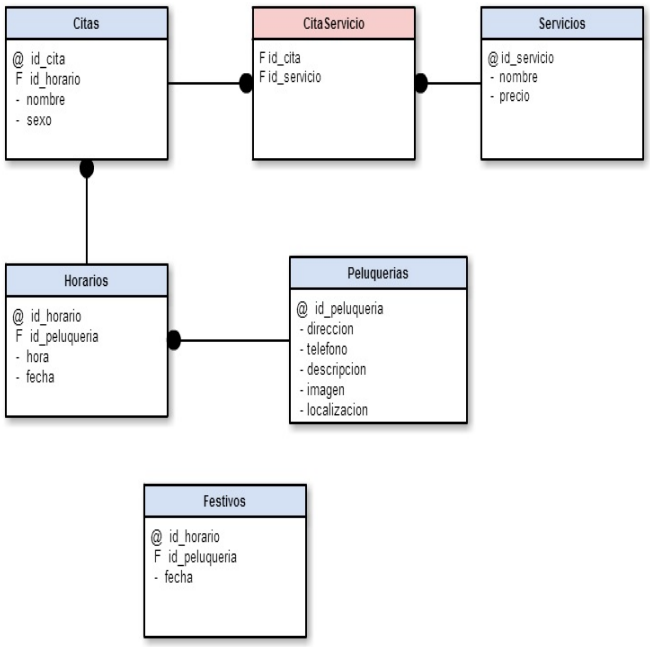

Figura 3.6: Diagrama de las tablas de la base de datos con sus campos

- **telefono:** es un campo de tipo *String* y almacena el teléfono de contacto de la peluquería.
- **descripcion:** es un campo de tipo *String* y almacena una breve descripción de la peluquería.
- **imagen:** es un campo de tipo *String* y almacena una URL donde se encuentra una imagen representativa de la peluquería.
- **localizacion:** es un campo tipo *String* y almacena las coordenadas geográficas exactas donde podemos encontrar la peluquería.

3.3.1.2. Entidad Horarios

Entidad que contiene los horarios que han reservado los usuarios de la *Aplicación Android para pedir cita previa en peluquerías*. Esta entidad cuenta con cuatro campos:

- **id_horario:** es un campo de tipo *String* y clave primaria. Almacena un identificador único para cada horario guardado.

- **id_peluqueria:** es un campo de tipo *String* y clave *Foránea [Peluquerias]*. Almacena el identificador de la peluquería en la que está ocupado el horario.
- **hora:** es un campo de tipo *String* y almacena el la hora que está ocupada.
- **fecha:** es un campo de tipo *String* y almacena la fecha que está ocupada.

3.3.1.3. Entidad Festivos

Entidad que contiene los dias festivos y días en los que no abre la peluquería. Esta entidad cuenta con tres campos:

- **id_horario:** es un campo de tipo *String* y clave primaria. Almacena un identificador único para cada horario guardado.
- **id_peluqueria:** es un campo de tipo *String* y clave *Foránea [Peluquerias]*. Almacena el identificador de la peluquería en la que está ocupado el horario.
- **fecha:** es un campo de tipo *String* y almacena la fecha festiva o en la que cierra la peluquería.

3.3.1.4. Entidad Citas

Entidad que contiene las citas que los usuarios han pedido a través de la *Aplicación Android para pedir cita previa en peluquerías*. Esta entidad cuenta con cuatro campos:

- **id_cita:** es un campo de tipo *String* y clave primaria. Almacena un identificador único para cada cita guardada.
- **id_horario:** es un campo de tipo *String* y clave *Foránea [Horarios]*. Almacena el identificador del horario que pertenece a dicha cita.
- **nombre:** es un campo de tipo *String* y almacena el nombre del usuario que ha pedido cita.
- **sexo:** es un campo de tipo *String* y almacena el sexo del usuario que ha pedido cita (este campo es necesario porque existe un peluquero para hombre y otro para mujer reservado para las peticiones a través de la aplicación, de tal forma que un hombre y una mujer pueden reservar el mismo horario).

3.3.1.5. Entidad Servicios

Entidad que contiene los servicios que se pueden contratar en la peluquería, por ejemplo: corte de pelo, tinte, corte de flecos, entre otros. Esta entidad cuenta con cuatro campos:

- **id_servicio:** es un campo de tipo *String* y clave primaria. Almacena un identificador único para cada cita guardada.
- **nombre:** es un campo de tipo *String* y almacena el nombre del servicio.
- **precio:** es un campo de tipo *String* y almacena el precio del servicio.

3.3.1.6. Entidad CitaServicio

Entidad que tiene como propósito principal relacionar cada cita con los servicios contratados. Sólo cuenta con dos campos, uno es **id_cita**, el identificador de la cita y clave *Foránea [Citas]* y el otro es **id_servicio**, el identificador del servicio y *Foránea [Servicios]*. Ambos campos son de tipo *String*.

3.3.2 Diseño de las clases e interfaces del modelo

El modelo de la *Aplicación Android para pedir cita previa en peluquerías*, como se puede ver en la figura 3.7, contará con una clase llamada *Modelo* y su interfaz *IModelo*. También contará con otras seis clases, una por cada tabla de la base de datos. En los siguientes apartados se describen cada una de las clases e interfases del modelo de la aplicación.

3.3.2.1. Clase *Modelo*

Esta clase implementa la interfaz *IModelo*, la cual define los siguientes métodos:

- **obtenerListaPeluquerias(): String**[]. Método que devuelve un vector con una lista de todas las peluquerías.
- **obtenerDescripcionPeluqueria(id_peluqueria: String): String**. Método que obtiene una descripción de una peluquería.
- **obtenerImagenPeluqueria(id_peluqueria: String): Bitmap** . Método que obtiene una imagen de una peluquería.

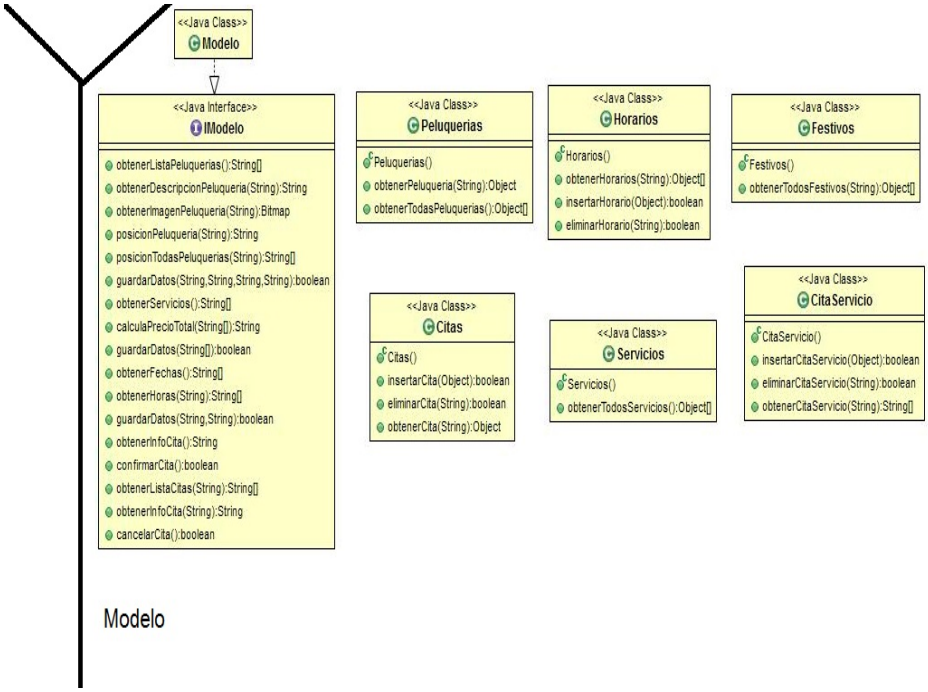

Figura 3.7: Clases e interfaces del modelo de la aplicación

- **posicionPeluqueria(id_peluqueria: String): String**. Método que obtiene la posición geográfica de una peluquería.
- **posicionTodasPeluquerias(id_peluqueria: String): String**[]. Método que obtiene un vector con la posición geográfica de todas las peluquerías.
- **guardarDatos(id_android: String, nombre: String, sexo: String, peluqueria: String): boolean** . Método encargado de guardar los datos en el modelo para posteriormente enviar al servidor.
- **guardarDatos(servicios: String[]): boolean** . Método encargado de guardar los datos en el modelo para posteriormente enviar al servidor.
- **guardarDatos(fecha: String, hora: String): boolean** . Método encargado de guardar ciertos datos en el modelo para posteriormente enviar al servidor.
- **obtenerServicios(): String**[] . Método que obtiene un vector con todos los servicios de la peluquería.
- **calculaPrecioTotal(servicios: String[]): String**. Método encargado de calcular el precio de los servicios contratados
- **obtenerFechas(): String**[]. Método encargado de obtener las fechas disponibles para poder pedir cita.

- **obtenerHoras(fecha: String): String**[]. Método encargado de obtener las horas disponibles en una fecha determinada.
- **obtenerInfoCita(): String**. Método encargado de obtener información de una cita.
- **confirmarCita(): boolean**. Método encargado de confirmar una cita, insertando dicha cita en la base de datos externa.
- **obtenerListaCitas(id_usuario: String): String**[]. Método encargado de obtener las citas que ha pedido un usuario determinado en la aplicación.
- **obtenerInfoCita(id_cita: String): String**. Método encargado de obtener información de una cita.
- **cancelarCita(): boolean**. Método encargado de cancelar una cita, eliminando dicha cita de la base de datos externa.

3.3.2.2. Clase *Peluquerias*

Esta clase tiene los siguientes métodos:

- **obtenerPeluqueria(id_peluqueria: String): Object**. Método que devuelve la peluquería de la base de datos que tiene el identificador que se le pasa por parámetros.
- **obtenerTodasPeluquerias(): Object**[]. Método que devuelve todas las peluquerías que hay guardadas en la base de datos.

3.3.2.3. Clase *Horarios*

Esta clase tiene los siguientes métodos:

- **obtenerHorarios(id_peluqueria: String): Object**[]. Método que devuelve los horarios de la base de datos que tienen el identificador de la peluquería que se le pasa por parámetros.
- **insertarHorario(horario: Object): boolean**. Método encargado de guardar un horario en la base de datos externa.
- **eliminarHorario(id_horario: String): boolean**. Método que elimina un horario de la base de datos externa.

3.3.2.4. Clase *Festivos*

Esta clase tiene los siguientes métodos:

- **obtenerFestivos(id_peluqueria: String): Object**[]. Método que devuelve los festivos de la base de datos que tienen el identificador de la peluquería que se le pasa por parámetros.

3.3.2.5. Clase *Citas*

Esta clase tiene los siguientes métodos:

- **insertarCita(cita: Object): boolean**. Método encargado de guardar una cita en la base de datos externa.
- **eliminarCita(id_cita: String): boolean**. Método que elimina una cita de la base de datos externa.
- **obtenerCita(id_cita: String): Object**. Método que devuelve la cita de la base de datos que tiene el identificador que se le pasa por parámetros.

3.3.2.6. Clase *Servicios*

Esta clase tiene los siguientes métodos:

- **obtenerTodosServicios(): Object**[]. Método que devuelve todos los servicios que hay guardados en la base de datos externa.

3.3.2.7. Clase *CitaServicio*

Esta clase tiene los siguientes métodos:

- **insertarCitaServicio(citaServicio: Object): boolean**. Método encargado de guardar una cita de un servicio (objeto *CitaServicio*) en la base de datos externa.
- **eliminarCitaServicio(id_cita: String): boolean**. Método que elimina una cita de un servicio (objeto *CitaServicio*) de la base de datos externa.
- **obtenerCitaServicio(id_cita: String): String**[]. Método que devuelve la cita de un servicio (objeto *CitaServicio*) de la base de datos que tiene el identificador que se le pasa por parámetros.

3.4 Diseño de las clases e interfaces del presentador

En esta sección se detallan las clases e interfaces que corresponden con la parte del presentador en el *MVP*, explicando cada método que aparece en la figura 3.8.

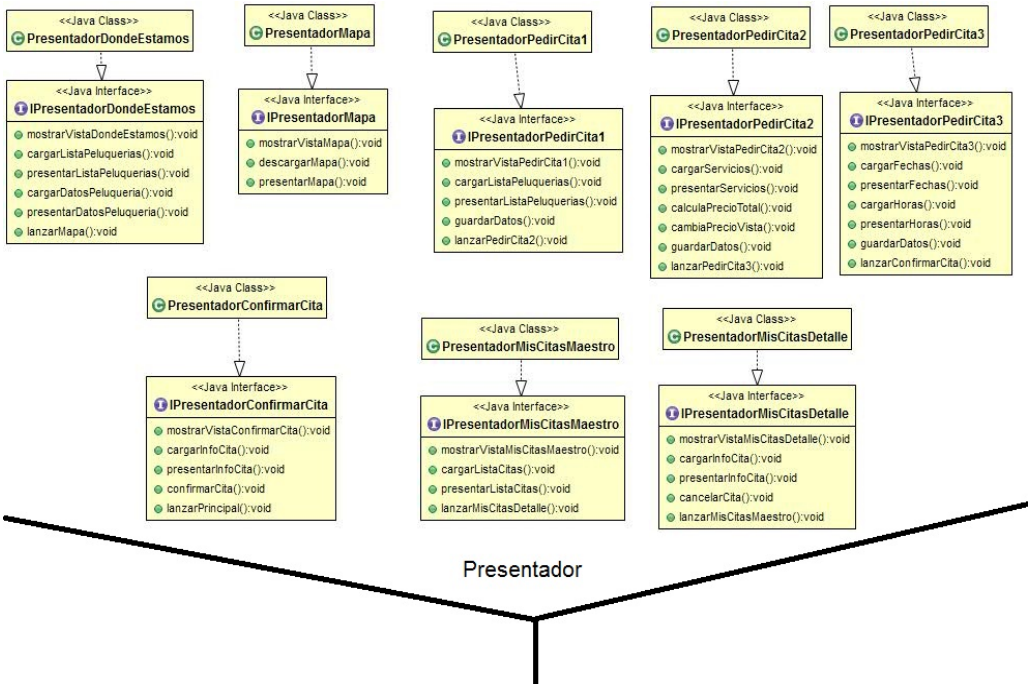

Figura 3.8: Clases del presentador de la aplicación con sus interfaces

3.4.1 Clase PresentadorDondeEstamos

Presentador correspondiente a la vista *DondeEstamosVistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorDondeEstamos*, la cual aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaDondeEstamos(): void**. Método que mostrará la vista *DondeEstamosVistaActivity*.
- **cargarListaPeluquerias(): void**. Método que pedirá al modelo la lista de peluquerías.
- **presentarListaPeluquerias(): void**. Método presentará en la vista la lista de peluquerías previamente recogida del modelo.

- **cargarDatosPeluqueria(): void**. Método que pedirá al modelo los datos de la peluquería seleccionada en la vista.
- **presentarDatosPeluqueria(): void**. Método que presentará en la vista los datos de la peluquería previamente cargados.
- **lanzarMapa(): void**. Método que pedirá al presentador de la vista *MapaVistaActivity* que la muestre en pantalla.

3.4.2 Clase PresentadorMapa

Presentador correspondiente a la vista *MapaVistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorMapa*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaMapa(): void**. Método que mostrará la vista *MapaVistaActivity*.
- **descargarMapa(): void**. Método que construye y descarga un mapa del servidor externo de Google.
- **presentarMapa(): void**. Método que presenta el mapa previamente construido y descargado.

3.4.3 Clase PresentadorPedirCita1

Presentador correspondiente a la vista *PedirCita1VistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorPedirCita1*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaPedirCita1(): void**. Método que mostrará la vista *PedirCita1VistaActivity*.
- **cargarListaPeluquerias(): void**. Método que pedirá al modelo la lista de peluquerías.
- **presentarListaPeluquerias(): void**. Método que presentará en la vista la lista de peluquerías previamente recogida del modelo.
- **guardarDatos(): void**. Método que guarda en el modelo los datos recogidos en la vista.
- **lanzarPedirCita2(): void**. Método que pedirá al presentador de la vista *PedirCita2VistaActivity* que la muestre por pantalla.

3.4.4 Clase PresentadorPedirCita2

Presentador correspondiente a la vista *PedirCita2VistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorPedirCita2*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaPedirCita2(): void**. Método que mostrará la vista *PedirCita2VistaActivity*.
- **cargarServicios(): void**. Método que carga del modelo los servicios de la peluquería.
- **presentarServicios(): void**. Método que presenta en la vista los servicios previamente cargados.
- **calculaPrecioTotal(): void**. Método que calcula el precio según los servicios seleccionados en la vista.
- **cambiaPrecioVista(): void**. Método actualiza en la vista el precio previamente calculado.
- **guardarDatos(): void**. Método que guarda en el modelo los datos recogidos en la vista.
- **lanzarPedirCita3(): void**. Método que pedirá al presentador de la vista *PedirCita3VistaActivity* que la muestre por pantalla.

3.4.5 Clase PresentadorPedirCita3

Presentador correspondiente a la vista *PedirCita3VistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorPedirCita3*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaPedirCita3(): void**. Método que mostrará la vista *PedirCita3VistaActivity*.
- **cargarFechas(): void**. Método que carga del modelo las fechas disponibles para pedir cita.
- **presentarFechas(): void**. Método que presenta en la vista las fechas disponibles.
- **cargarHoras(): void**. Método que carga del modelo las horas disponibles según la fecha seleccionada en la vista.

- **presentarHoras(): void**. Método que presenta en la vista las horas disponibles previamente cargadas.
- **guardarDatos(): void**. Método que guarda en el modelo los datos recogidos en la vista.
- **lanzarConfirmarCita(): void**. Método que pedirá al presentador de la vista *ConfirmarCitaVistaActivity* que la muestre por pantalla.

3.4.6 Clase PresentadorConfirmarCita

Presentador correspondiente a la vista *ConfirmarCitaVistaActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorConfirmarCita*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaConfirmarCita(): void**. Método que mostrará la vista *ConfirmarCitaVistaActivity*.
- **cargarInfoCita(): void**. Método que carga del modelo la información de la cita.
- **presentarInfoCita(): void**. Método que presenta en la vista la información de la cita previamente cargada del modelo.
- **confirmarCita(): void**. Método que guarda en la base de datos externa la cita a través del modelo.
- **lanzarPrincipal(): void**. Método que pedirá al presentador principal que muestre la vista principal de la aplicación.

3.4.7 Clase PresentadorMisCitasMaestro

Presentador correspondiente a la vista *MisCitasVistaMaestroActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorMisCitasMaestro*, la cual aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaMisCitasMaestro(): void**. Método que mostrará la vista *MisCitasVistaMaestroActivity*.
- **cargarListaCitas(): void**. Método que carga del modelo una lista con las citas que ha pedido el usuario de la aplicacion.

- **presentarListaCitas(): void**. Método que presenta en pantalla la lista con las citas previamente cargadas del modelo.
- **lanzarMisCitasDetalle(): void**. Método que pedirá al presentador de la vista *MisCitasVistaDetalleActivity* que la muestre por pantalla.
- **lanzarPrincipal(): void**. Método que pedirá al presentador principal que muestre la vista principal de la aplicación.

3.4.8 Clase PresentadorMisCitasDetalle

Presentador correspondiente a la vista *MisCitasVistaDetalleActivity*, que es el encargado de mostrar dicha vista, actualizarla y presentar la información pertinente en ésta. Esta clase implementa la interfaz *IPresentadorMisCitasDetalle*, que aparece representada en la figura 3.8 con los siguientes métodos:

- **mostrarVistaMisCitasDetalle(): void**. Método que mostrará la vista *MisCitasVistaDetalleActivity*.
- **cargarInfoCita(): void**. Método que carga del modelo la información de la cita.
- **presentarInfoCita(): void**. Método que presenta en la vista la información de la cita previamente cargada del modelo.
- **cancelarCita(): void**. Método que borra de la base de datos externa la cita a través del modelo.
- **lanzarMisCitasMaestro(): void**. Método que pedirá al presentador de la vista *MisCitasVistaMaestroActivity* que la muestre por pantalla.

3.5 Adecuación del diseño a Android

Android es una plataforma para dispositivos móviles que contiene una pila de software donde se incluye un sistema operativo, middleware y aplicaciones básicas para el usuario. En la figura 3.9 se observan las distintas capas que componen Android. Cada una de estas capas utiliza servicios ofrecidos por las anteriores, y ofrece a su vez los suyos propios a las capas de niveles superiores [13]. Las capas se definen brevemente a continuación:

- **Aplicaciones:** contiene, tanto las incluidas por defecto de Android como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las API y bibliotecas de los niveles anteriores.

- **Framework de Aplicaciones:**. representa el conjunto de herramientas de desarrollo de cualquier aplicación. Toda aplicación que se desarrolle para Android utilizan el mismo conjunto de API y el mismo framework, representado por este nivel. Entre las API más importantes, se pueden encontrar las siguientes:
  - **Activity Manager:** conjunto de API que gestiona el ciclo de vida de las aplicaciones en Android.
  - **Content Provider:** permite a cualquier aplicación compartir sus datos con las demás aplicaciones de Android.
  - **View System:** proporciona un gran número de elementos para poder construir interfaces de usuario (GUI), como listas, mosaicos, botones, check boxes, tamaño de ventanas, control de las interfaces mediante teclado, entre otros.
  - **Location Manager:** posibilita a las aplicaciones la obtención de información de localización y posicionamiento.
- **Bibliotecas:** éstas proporcionan a Android la mayor parte de sus capacidades más características. Junto al núcleo basado en Linux, estas bibliotecas constituyen el corazón de Android. Entre las bibliotecas más importantes, se pueden encontrar las siguientes:
  - **OpenGL/SL y SGL:** Representan las bibliotecas gráficas. OpenGL/SL maneja gráficos en 3D y SGL proporciona gráficos en 2D.
  - **Librería SQLite:** creación y gestión de bases de datos relacionales.
- **Android Runtime:** al mismo nivel que las bibliotecas de Android se sitúa el entorno de ejecución. Éste lo constituyen las *Core Libraries*, que son bibliotecas con multitud de clases Java y la máquina virtual Dalvik.
- **Núcleo Linux:** Android utiliza el núcleo de Linux como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los *drivers* necesarios para que cualquier componente hardware pueda ser utilizado mediante las llamadas correspondientes. Siempre que un fabricante incluye un nuevo elemento de hardware, lo primero que se debe realizar para que pueda ser utilizado desde Android es crear las bibliotecas de control o drivers necesarios dentro de este kernel de Linux embebido en el propio Android.

3.5.1 Adecuación de la arquitectura MVP

Teniendo en cuenta la arquitectura *MVP* y con el fin de que la aplicación a desarrollar esté, a nivel de programación, lo más desacoplada posible (con el fin de

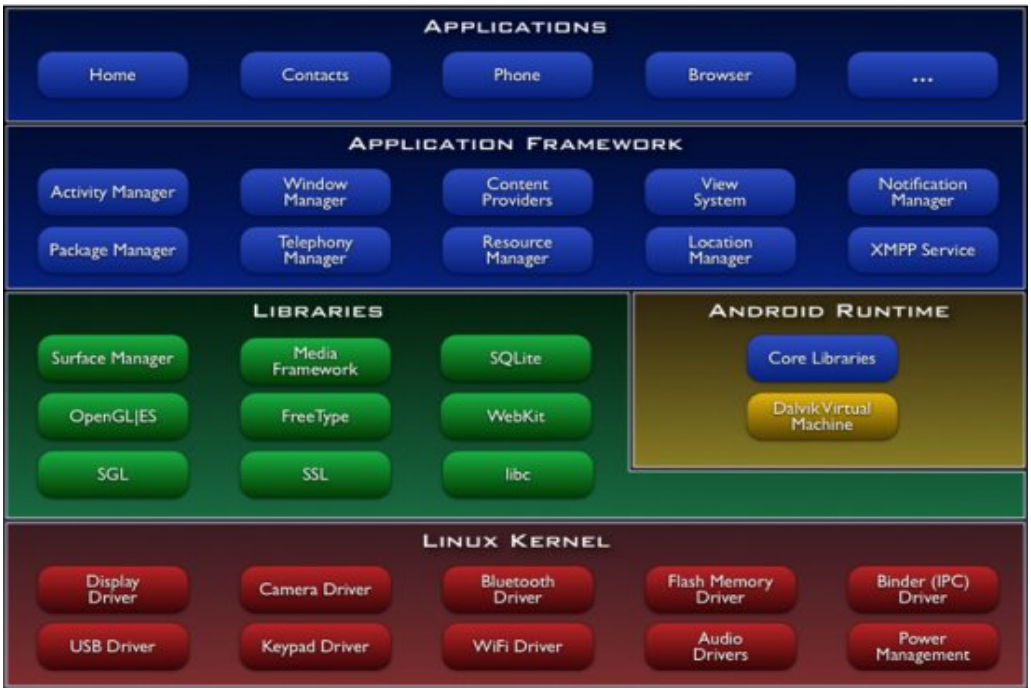

Figura 3.9: Arquitectura del sistema operativo Android

poder actualizar cualquier parte del sistema sin afectar al resto), se diseña la codificación de forma que las clases e interfaces de la *Vista*, el *Modelo* y el *Presentador* se almacenen en los paquetes vista, modelo y presentador, respectivamente. Las clases de la vista serán ventana de la interfaz de usuario, que en Android corresponden con clases que extienden de la clase Activity o de alguna de sus subclases [14]. Por otro lado, en la arquitectura *MVP* tradicional, el presentador es el punto de entrada a la aplicación y éste es el que crea a la vista y al modelo. Sin embargo, en Android, el punto de entrada a la aplicación es la vista principal y por tanto, la arquitectura *MVP*, tal y como se conoce, no puede ser implementada, por lo que hay que realizar ciertas modificaciones. La modificación más importante es el uso de una clase que conozca a todas las componentes de la aplicación y que se encargue del control de la navegación en la misma. Esta clase recibe el nombre de *AppMediador* y deriva de la clase *Application* [15] de Android (se almacena fuera de los paquetes indicados anteriormente).

Así, cuando se lanza la aplicación, se entra en la vista principal y ésta se encargará de obtener el objeto aplicación (es decir, el objeto de tipo *AppMediador*) y de indicarle a éste, quién es la vista principal. Cada vista de la aplicación, cuando sea cargada, debe indicar al objeto *AppMediador* quién es para que sus presentadores lo sepan (cuando necesiten acceder a sus vistas). Asimismo, cuando una vista tenga que delegar en su presentador para atender los eventos de usuario, le pedirá al objeto *AppMediador* que le indiqué qué objeto es su presentador. Por otro lado, el presentador de la vista principal se encargará de crear el objeto modelo y de indicar

al objeto *AppMediador* quién es este modelo, para que cualquier otro presentador pueda acceder a él.

Para el almacenamiento de la información en la *Aplicación Android para pedir cita previa en peluquerías* se va utilizar la plataforma Parse [16]. *Parse* es una plataforma de servicios online creada para facilitar la tarea de creación de un *back-end* a una aplicación móvil, entendiendo por *back-end* a los sistemas e infraestructuras necesarios para que los datos de una aplicación concreta sean accesibles desde la web.

Parse ofrece una infraestructura a partir de la cual se puede empezar a desarrollar una aplicación móvil (también se pueden desarrollar otro tipos de aplicaciones), de forma gratuita dependiendo del tráfico de consultas que exista entre la aplicación y los servicios prestados por *Parse*, en este caso en particular al no superar el millón de consultas por mes no será necesario realizar ningún pago a la organización.

Gracias a la infraestructura de *Parse* y a su sistema de administración, se reduce el tiempo de desarrollo, al evitar tener que crear una base de datos externa en un servidor y realizar consultas a él directamente, o crear un servicio REST para poder hacer consultas de forma remota. *Parse* ofrece un administrador en el que se crea el modelo de datos (no utiliza base de datos SQL) y una API de alto nivel que se añade fácilmente al SDK de Android facilitando la consultas al modelo de datos.

Los tipos de datos que soporta el modelo de datos que proporciona *Parse* son los son siguientes: *String, Number, Boolean, Date, File, Geopoint, Array, Object, Pointer, Relation.* En el caso de la aplicación objeto de este trabajo, los campos de las tablas de la base de datos serán de alguno de los tipos anteriores (en el que se adapte mejor según el tipo elegido).

3.5.2 Identificación de los patrones usados

Los patrones de diseño se utilizan para resolver problemas comunes de ingeniería. Cuando se usa un patrón de diseño para resolver un problema, se adapta el patrón a las necesidades específicas de ese problema.

Android ofrece en su web un apartado de diseño [17] en el que ofrece patrones para distintas tareas. De los diferentes patrones que ofrece, en la *Aplicación Android para pedir cita previa en peluquerías* se usaran los siguientes:

- **Action Bar.** Barra superior presente en cualquier vista de la aplicación. En la que puede aparecer el nombre de la aplicación o vista en la que se encuentre, contiene botones para las acciones mas importantes y oculta en un botón las acciones menos importantes o que se utilizan en menor medida, tal y como se observa en la figura 3.10.

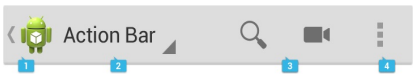

Figura 3.10: Action Bar

- **Confirmación y Reconocimiento.** En los momentos en los que se invoca alguna acción, como por ejemplo en el momento en el que el usuario selecciona el botón de salir de la aplicación, es una buena idea ofrecer al usuario la opción de confirmar la acción, por si le ha dado de forma errónea. En el caso de la *Aplicación Android para pedir cita previa en peluquerías* habrá confirmación cada vez que el usuario quiera salir de la aplicación y también habrá reconocimiento cuando el usuario, confirme o cancele cita.
- **Preferencias.** Se ofrece al usuario un lugar en su aplicación donde indica sus preferencias con la forma en la que su aplicación debe comportarse. Esto beneficia a los usuarios debido a que no es necesario que se les interrumpa con las mismas preguntas una y otra vez cuando se presentan ciertas situaciones. Los ajustes predeterminan lo que siempre va a pasar en esas situaciones.
- **Ayuda.** Aunque se debe hacer siempre una aplicación en la que el uso de la ayuda sea innecesario, siempre deber existir una apartado de ayuda en el que el usuario pueda resolver sus dudas y aprender más sobre la aplicación.

En cuanto a los patrones de diseño de software, la *Aplicación Android para pedir cita previa en peluquerías* usará:

- **Singleton:** la clase AppMediador implementa este patrón de forma que sólo existe un objeto de este tipo en la aplicación (no permiten la creación de más de un objeto de este tipo).
- **Delegado:** las vistas de la aplicación delegan el tratamiento de las acciones del usuario (por ejemplo, la selección de un determinado botón), a sus presentadores. Así, los presentadores realizarán las operaciones oportunas en nombre de sus vistas.
- **Observador:** los presentadores de la aplicación deben observar al modelo, de forma que cuando éste termine de realizar el acceso a la información, los presentadores deben saberlo. En Android, para realizar este proceso, se usan las notificaciones broadcast (o lo que es lo mismo, un objeto de tipo *BroadcastReceiver* [18]).
- **Maestro-Detalle:** existen dos vistas, la vista maestro con un listado de objetos y la vista detalle con información del objeto que se ha seleccionado previamente en la vista maestro. En la *Aplicación Android para pedir cita previa en peluquerías* este patrón se ve reflejado claramente en el caso de uso *Mostrar Citas*.