Capítulo 4

MODELO DE IMPLEMENTACIÓN

4.1 Introducción

El *Modelo de implementación* utiliza el resultado del *Modelo de diseño* para generar el código final en el lenguaje de programación elegido [10]. Aunque el diseño de objetos es independiente del lenguaje de programación final, todos los lenguajes tienen particularidades que deben adaptarse durante el proceso de implementación. La elección del lenguaje influye en el diseño, pero el diseño no debe depender de los detalles del lenguaje, es decir, un cambio de lenguaje no puede hacer que el diseño cambie.

Asimismo, y como ya se ha comentado en capítulos anteriores, el diseño de la aplicación utiliza la arquitectura *Modelo-Vista-Presentador* (MVP) adaptada al lenguaje elegido. En esta adaptación, se incorpora una nueva clase, llamada *AppMediador* (introducida en el *Modelo de Diseño*) que va a funcionar como punto de enlace de las distintas partes de la arquitectura. Así, el *AppMediador*, que representa a la clase *Application* (aplicación) se encargará de:

- Definir los presentadores, vistas y modelo de la arquitectura y los métodos *accessor* de éstos.
- Definir la navegación en la aplicación, es decir, qué vistas (de la interfaz de usuario) se deben lanzar cuando se considere oportuno (normalmente, ante peticiones del usuario).
- Definir las constantes correspondientes a los avisos de notificación, enviadas desde el modelo cuando se haya terminado de realizar una determinada acción.
- Definir las constantes correspondientes a los datos comunicados entre vistas, o recuperados desde el modelo, entre otros, ya que en Android los datos se comunican usando pares *clave,valor*.
- Definir los métodos propios de Java para Android correspondientes al: lanzamiento de actividades, creación de servicios, registros de receptores *broadcast*, envíos de notificaciones *broadcast*, entre otros.

Es importante comentar que para que la aplicación desarrollada utilice este objeto *AppMediador*, en el archivo *AndroidManifest* hay que indicar que el nombre de la aplicación corresponde con este archivo. Para ello, hay que indicar en este archivo, dentro de la etiqueta *application*, lo siguiente:

```
<application
   android:name="rutaHastaLaClase.AppMediador"
   ...>
</aplication>
```

En la siguientes secciones se especificarán los cambios realizados en la implementación del diseño, empezando por la descripción del *AppMediador* para esta aplicación y siguiendo con las modificaciones realizadas en ciertas clases de los tres paquetes definidos: *vista*, *presentador* y *modelo*, debido a la implementación en Java para Android y a la arquitectura elegida.

4.2 Clase AppMediador

El *AppMediador* contiene una variable por cada una de las interfaces del presentador, la vista y el modelo de la aplicación *Aplicación Android para pedir cita previa en peluquerías*.

Por otro lado, se definen unas constantes de petición y notificación (*public* y *static*) que serán las claves de los *Intent* (*clave, valor*). Se utilizarán en los objetos *BroadcastReceiver* y en los métodos *sendBroadcast* para notificar cuándo ha finalizado un evento en la aplicación (por ejemplo, el modelo notifica cuándo termina de insertar datos en la base de datos).

Asimismo, se implementan:

- Los métodos de creación y eliminación de presentadores: "getPresentadorXXX" y "removePresentadorXXX" (donde *XXX* corresponde al nombre de un presentador), para cada una de las variables creadas con anterioridad y que representan a los presentadores introducidos en el *Modelo de diseño*.
- Los métodos *accessor* de las vistas definidas en el *Modelo de requisitos*: "getVistaXXX" y "setVistaXXX" (donde *XXX* corresponde al nombre de una vista). Estos métodos son utilizados para obtener o iniciar las variables de las actividades (*Activity* de Android).
- Los métodos *accessor* del modelo: "getModelo" y "setModelo".
- Los métodos de navegación.
- Los métodos de uso de Java para Android.

- El método "onCreate", que inicia todas las variables de los presentadores a *null* e indica que la clase *AppMediador* es un *singleton*.

La vista principal de la aplicación, que corresponde con la clase *PrincipalVistaActivity*, crea el objeto *AppMediador* con el método "getApplication", para que el resto de clases del proyecto puedan usarlo. Asimismo, el presentador principal, que corresponde con la clase *PresentadorPrincipal*, crea al modelo, que se encarga de manejar el almacenamiento de los datos.

4.3 Paquete vista

En el paquete vista se han implementado las clases y métodos que se observan en la imagen 4.1. A continuación se identifican las diferencias entre las clases e interfaces de la vista propuestas en el *Modelo de requisitos* y las finalmente implementadas. Así, los cambios son los siguientes:

- En todas las vistas que interactúan con la base de datos externa se han añadido los métodos:
  - **void mostrarAlerta(String titulo)**. Método que muestra en la vista una alerta con el texto que se le pasa por parámetros.
  - **void mostrarProgreso(String mensaje)**. Método que muestra una barra de progreso para indicar que la aplicación está realizando alguna operación.
  - **void eliminarProgreso()**. Método que elimina la barra de progreso de la vista.
- En Andrid se requiere que se añadan todas las clases de las vistas mediante los siguientes métodos:
  - **void onCreate(Bundle savedInstanceState)**. Método que inicia la actividad. Sólo se ejecuta una vez.
  - **void onStart()**. Método que se ejecuta justo después del *onCreate* o cuando se vuelve a la actividad. La actividad se vuelve visible para el usuario cuando se llama a este método.
  - **boolean onCreateOptionsMenu(Menu menu)**. Inicia el contenido del menú de opciones de la actividad.
  - **boolean onOptionsItemSelected(MenuItem item)**. Método que se llama cada vez que se selecciona un elemento en el menú.
  - **void onClick(View v)**. Método de la interfaz *OnClickListener* que se invoca cuando se presiona sobre un botón de una vista, para realizar las acciones oportunas. Se ha añadido este método para aquellas vistas que lo requieran.

4.3.1 Clase Clase MapaVistaActivity e interfaz IVistaMapa

Al utilizar la API de Google Maps para Android los cambios que se hacen en el presentador sobre el mapa se actualizan directamente en la vista, por ello se ha suprimido el siguiente método al quedar inutilizado:

- **setMapa(mapa: Object)**

4.3.2 Clase PedirCita1VistaActivity e interfaz IVistaPedirCita1

Se ha añadido esta vista un campo de texto en el que se introduce el teléfono móvil por cuestiones de funcionalidad. Por ello ha sido necesario añadir el siguiente método:

- **String getTextoTelefono()**. Devuelve el teléfono introducido en la vista.

4.3.3 Clase PedirCita2VistaActivity e interfaz IVistaPedirCita2

Se ha decidido que el precio de los servicios se cargue del servidor externo y en consecuencia en la vista se añaden los siguientes métodos:

- **void setPrecioCorteDePelo(String precio)**. Actualiza el precio del corte de pelo en la vista.
- **void setPrecioManicura(String precio)**. Actualiza el precio de la manicura en la vista.
- **void setPrecioPedicura(String precio)**. Actualiza el precio de la pedicura en la vista.
- **void setPrecioMechas(String precio)**. Actualiza el precio de las mechas en la vista.
- **void setPrecioTinte(String precio)**. Actualiza el precio del tinte en la vista.
- **void setPrecioCorteDeFlecos(String precio)**. Actualiza el precio del corte de flecos en la vista.

4.3.4 Clase MisCitasVistaMaestroActivity e interfaz IVistaMisCitasMaestro

Para mostrar el listado de citas del usuario es necesario identificarlo y se hace mediante número de teléfono, se ha añadido el siguiente método en la vista para recoger el teléfono del usuario:

- **String getTelefono()**. Devuelve el teléfono introducido en la vista.

4.4 Paquete presentador

En el paquete presentador se han implementado las clases y métodos que se observan en la imagen 4.2. A continuación se identifican las diferencias entre las clases e interfaces del presentador propuestas en el *Modelo de diseño* y las finalmente implementadas. Así, los cambios son los siguientes:

- En los presentadores que necesitan datos del servidor externo se ha añadido una variable privada de la clase *BroadcastReceiver* para detectar los avisos desde el modelo cuando termina de descargar, insertar o eliminar datos.

4.4.1 Clase PresentadorMapa e interfaz IPresentadorMapa

Se ha suprimido el método, **descargarMapa(): void**, porque al utilizar la API de Google Maps, se realiza este método de manera autónoma.

Se han añadido los siguientes métodos:

- **void presentarMapaTodasPeluquerias()**. Presenta un mapa con todas las peluquerías.
- **void presentarMapaPeluqueriaCercana()**. Presenta un mapa con la peluquería más cercana.

4.5 Paquete modelo

En el paquete modelo se han implementado las clases y métodos que se observan en la imagen 4.3. A continuación se identifican las diferencias entre las clases e interfaces del modelo propuestas en el *Modelo de diseño* y las finalmente implementadas. Así, se han añadido las siguientes clases:

- **DatosCita**. Clase necesaria para trabajar con bases de datos externa. Esta clase representa un objeto con los datos de la cita.
- **DatosCitaServicio**. Clase necesaria para trabajar con bases de datos externa. Esta clase representa un objeto con las relaciones de las citas con los servicios.
- **DatosHorario**. Clase necesaria para trabajar con bases de datos externa. Esta clase representa un objeto con los horarios de las citas.
- **DatosPeluqueria**. Clase necesaria para trabajar con bases de datos externa. Esta clase representa un objeto con los datos de la peluquerías.
- **DatosServicios**. Clase necesaria para trabajar con bases de datos externa. Esta clase representa un objeto con los datos de los servicios.

4.5.1 Clase Modelo e interfaz IModelo

Se modifican los métodos del *Modelo* que interactúan con el servidor externo, ahora no retornan ninguna variable, son de tipo void. Los datos del modelo ahora se envían utilizando notificaciones broadcast con objetos de tipo *BroadcastReceiver*.

Se han añadido los siguientes métodos:

- **void setCitaSeleccionada(String id_cita)**. Guarda la cita seleccionada.
- **String getCitaSeleccionada()**. Obtiene la cita seleccionada.
- **void setPeluqueriaSeleccionada(String id_peluqueria)**. Guarda la peluquería seleccionada.
- **String getPeluqueriaSeleccionada()**. Obtiene la peluquería seleccionada.
- **void setTelefono(String telefono)**. Guarda el teléfono del usuario.
- **String getTelefono()**. Obtiene el teléfono del usuario
- **void obtenerDatosPeluquerias()**. Obtiene los datos de las peluquerías guardadas en el servidor externo.

Figura 4.1: Clases del paquete vista

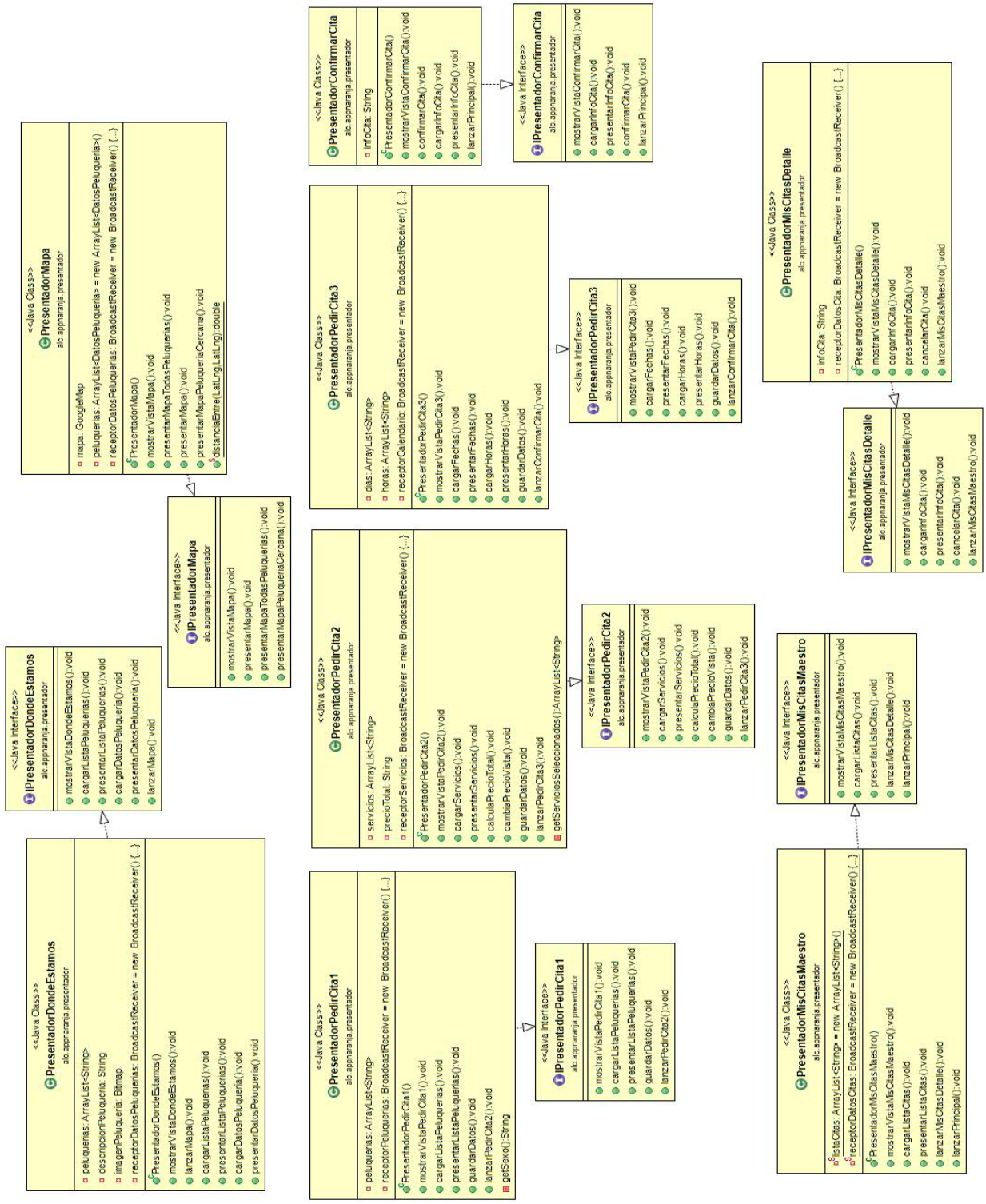

Figura 4.2: Clases del paquete presentador

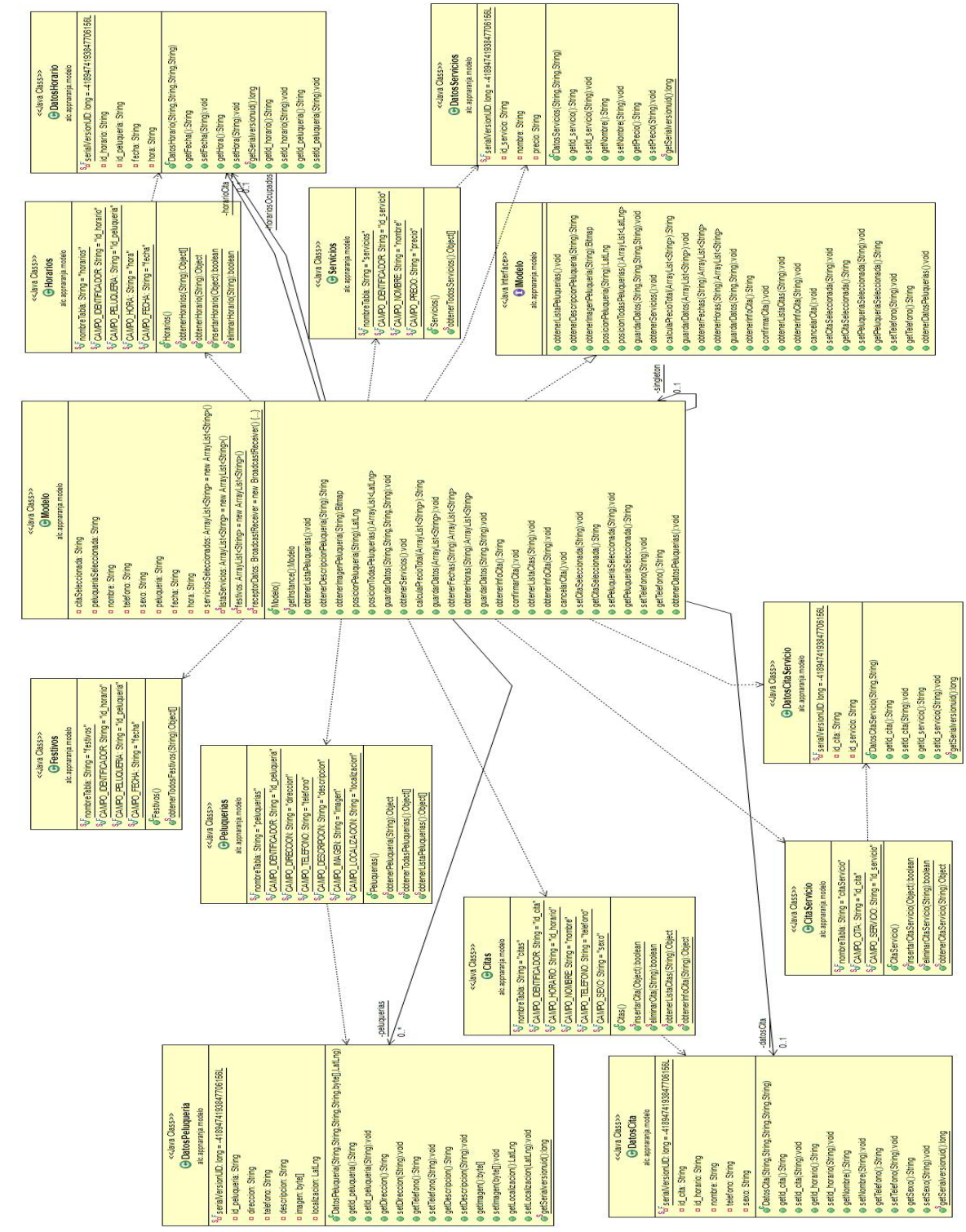

Figura 4.3: Clases del paquete modelo