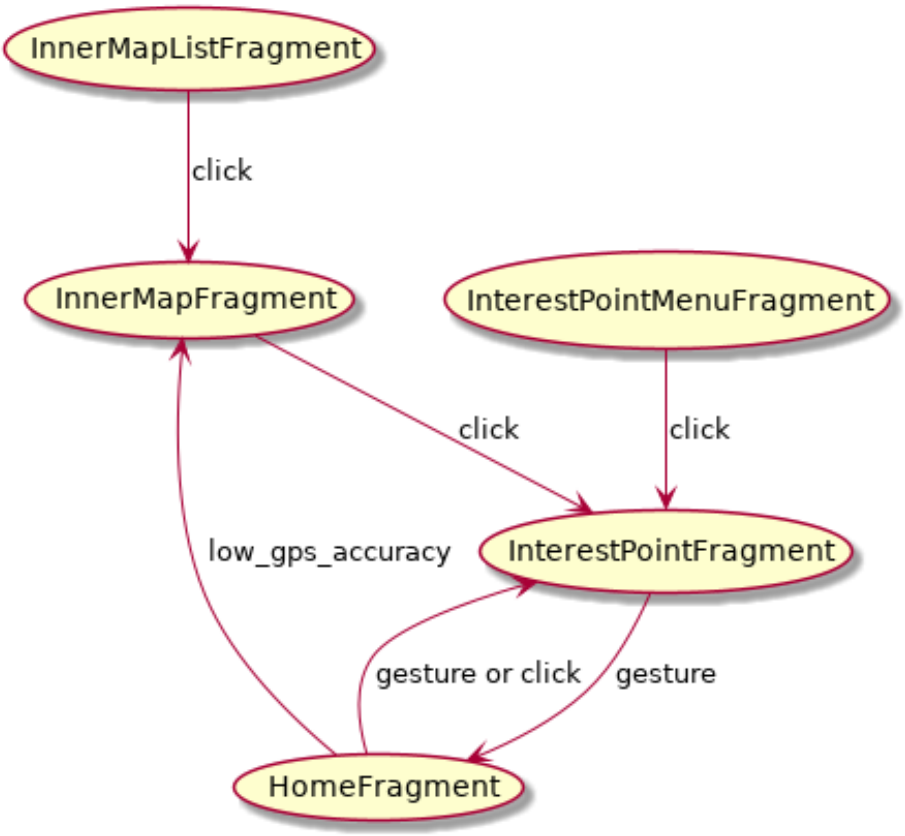
Estructura del proyecto

La aplicación `Alamabra1925` está implementada como un proyecto `Android` en `Kotlin`. El proyecto se encuentra estructurado en una única actividad `MainActivity` y una serie de fragmentos, cada uno de ellos encargado de una faceta de la aplicación. Estos son:

- `HomeFragment`.
- `InnerMapListFragment`.
- `InnerMapFragment`.
- `InterestPointFragment`.
- `InterestPointMenuFragment`.
- `GameFragment`.

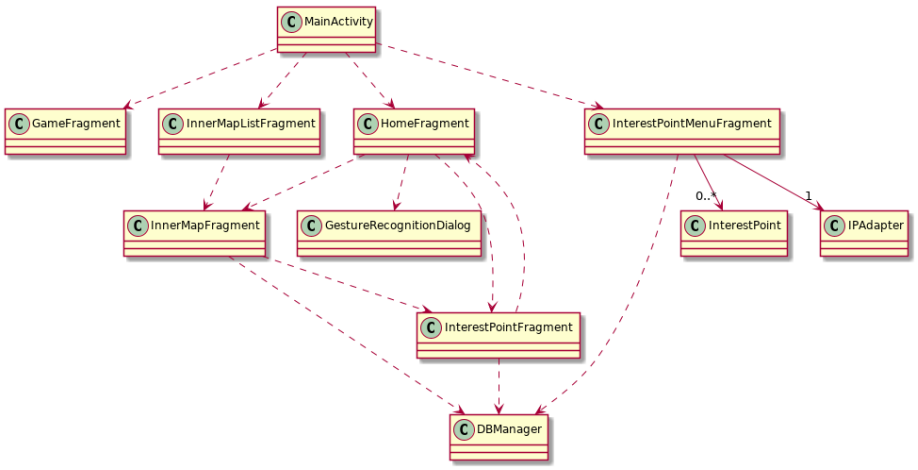
Todos los fragmentos y las interacciones entre ellos se llevan a cabo utilizando un "controlador de navegación" (`navController`). Este nos permite navegar entre los fragmentos desde la barra de navegación lateral y entre ellos utilizando `actions`. Estas acciones se pueden comprobar en el fichero `res/navigation/mobileNavigation.xml`.

El siguiente diagrama muestra información equivalente.



Desde esta barra lateral podemos acceder a `HomeFragment` (Map), `InnerMapListFragment` (Inside Maps) y `GameFragment` (Game).

Ahora mostramos el diagrama de clases de la aplicación, sin tener en cuenta atributos o métodos de cada una de ellas. Debido a la estructura con fragmentos del proyecto, las clases no contienen instancias de otras clases, salvo por `InterestPointMenuFragment`. Por esto, hemos decidido simbolizar en el diagrama las comuninaciones entre las clases (mediante líneas discontinuas).



Fragmentos

HomeFragment

Este es el fragmento principal de la aplicación, en el que se implementa el mapa de la aplicación y todas las interacciones con él. En primer lugar se declara un mapa utilizando la librería `osmdroid`, donde situamos al usuario. Podriamos haber optado por utilizar la API de GoogleMaps, pero vimos más conveniente utilizar la alternativa *open source*. Para hacerlo seguimos los siguientes pasos dentro de la función `onCreateView`:

- Tomamos la vista de "openstreetmap".
- Añadimos el compás al mapa. Para ello utilizamos `CompassOverlay`.
- Añadimos la localización actual del usuario con `MyLocationNewOverlay`.
- Tomamos el controlador del mapa y ajustamos ciertos parámetros como el zoom inicial, la longitud de la barra de escala y el seguimiento.

A continuación añadimos los `markers` de los distintos puntos de interés importados de la base de datos. Solo mostramos aquellos con `tipo = 0` ya que en la base de datos se guardan puntos de interés externos (con `tipo = 0`) y puntos de interés internos (con `tipo > 0`).

Para añadirlos se utiliza la funcion `addMarker`, a la que se le pasa la **longitud** y **latitud** del punto y su identificador en la base de datos. En la función se crea un punto en el mapa y se le añade un `ClickListener`, el cual cambia de vista al pulsarse mostrando la información del punto de interés.

Por otro lado se añade también un `locationListener`, que se actualiza cada segundo o cuando el usuario se ha desplazado un metro. Cuando el *listener* detecta que la posición cambia llama a la función `onLocationChanged`.

Dentro de esta última se comprueba si la señal de GPS es mayor que un cierto

`UMBRAL`, para comprobar si se ha producido una pérdida de señal GPS. En este caso se puede considerar que el usuario está entrando a un monumento, por lo que se verifica si se encuentra dentro de los edificios de mapa interior. Para ello, la función `nearDoor` realiza lo siguiente:

- Para cada uno de los monumentos, buscamos un cuadrado en el espacio que encierre toda la estructura.
- Aprovechamos que en los primeros momentos de andentrarse en la estructura la señal de GPS aún no desaparece para calcular la posición del usuario.
- Comprobamos si el usuario se encuentra dentro de alguno de los cuadrados definidos antes (vemos la latitud y la longitud como las coordenadas de un punto en un plano).

Cuando esto ocurre, se desactiva el `locationlistener` y se cambia de vista al fragmento del mapa interior con el identificador correspondiente.

InnerMapFragment

En este fragmento se implementan los mapas internos junto con sus puntos de interés. Podemos llegar a él desde `HomeFragment` o `InnerMapListFragment`, y en ambos casos será necesario añadir un identificador `id` al `bundle` de la navegación, que simboliza cuál de los mapas interiores queremos cargar.

Para ello utilizamos

```
val mid = arguments!!.get("id")
```

También lo utilizaremos para saber cuál de los conjuntos de puntos de interés debemos cargar en la vista del mapa.

Para añadir estos puntos utilizamos la función `addFloatingButton`, que nos permitirá utilizar los valores presentes en la base de datos para colocarlo correctamente en la vista. Para ello seguiremos los siguientes pasos:

- Crear un `floatingActionButton`.
- Crear un `relativeLayout` sobre el que añadir los márgenes.
- Establecer el layout al botón.
- Añadir ciertas características del botón, como el icono, el tamaño y los colores.
- Establecer el `Listener` de pulsación para lanzar la vista del punto de interés correspondiente.

InnerMapListFragment

En este fragmento alojaremos una lista de `String`, donde cada uno de ellos simboliza un mapa interior de la Alhambra. Para ello utilizamos una `listView` y un `ArrayAdapter`. En cada uno de los elementos de la lista sobrecargamos la pulsación para que añada al `bundle` el identificador correspondiente.

InterestPointFragment

Este fragmento nos permite visualizar toda la información acerca de un punto de interés concreto de nuestra visita. En el método `onCreateView` del fragmento, esperamos un `Bundle` con un identificador que nos permita conocer de qué fragmento se debe mostrar la información.

Con esto utilizamos la base de datos para rellenar los `textView` apropiados.

En este fragmento, utilizamos la detección de una "pinza" de 3 dedos para volver al mapa. Para hacer esto, seguimos los siguientes pasos.

- Declaramos el objeto *listener*.

```
private val mOnTouchListener = object : View.OnTouchListener {}
```

- Sobrecargamos el método `onTouch` con el siguiente código con tres eventos.
  - Cuando detectamos 3 dedos, tomamos sus alturas iniciales.

```
val action = event!!.actionMasked
if (event.pointerCount >= 3 && action ==
    MotionEvent.ACTION_POINTER_DOWN) {
        ini0 = event.getY(0)
        ini1 = event.getY(1)
        ini2 = event.getY(2)
        reg = true
}
```

  - Cuando soltamos los dedos, comparamos si se ha realizado la accion de pinza.

```
if (action == ACTION_UP && reg) {
    reg = false
    if (
        (ini0 < fin0  && ini1 < fin1  && ini2 > fin2) ||
        (ini0 > fin0  && ini1 < fin1  && ini2 < fin2) ||
        (ini0 < fin0  && ini1 > fin1  && ini2 < fin2)
        )
        {
            findNavController().navigate
            (R.id.action_nav_ip_to_nav_home, null)

        }
    }
```

+ Si se produce movimiento, registramos la posicion de los 3 dedos.

```
if (action == MotionEvent.ACTION_MOVE && reg
&& event.pointerCount >= 3) {
    fin0 = event.getY(0)
    fin1 = event.getY(1)
```

```
            fin2 = event.getY(2)
        }
```

InterestPointMenuFragment

A este fragmento se puede acceder desde la barra de navegación lateral bajo el nombre de `Puntos de interés`. Aquí obtendremos una lista de todos los puntos de interés de los que se tiene información en la aplicación. Para ello utilizamos un adaptador `IPAdapter` junto con la base de datos.

Cada uno de los puntos de la lista tiene sobrecargado el *listener* de pulsación, haciendo que sea posible ir a la vista detallada de cada uno de ellos.

En el método `onCreateView` hacemos lo siguiente.

```
val mainList = root.findViewById<ListView>(R.id.list_view)
listAdapter = IPAdapter(this.context!!, list)
mainList.adapter = listAdapter

mainList.onItemClickListener =
    AdapterView.OnItemClickListener { _, _, position, _ ->
        val bundle = bundleOf("id" to list[position].id)
        findNavController().navigate
            (R.id.action_nav_ip_menu_to_nav_ip, bundle)
}

loadQueryAll()
```

- Inicializamos la variable correspondiente a la vista de la lista.
- Inicializamos nuestro adaptador y se lo asignamos a la lista.
- Sobrecargamos el *listener* de pulsación de forma que utilice el controlador de navegación para cargar el fragmento del punto de interés correspondiente.
- Utilizamos `loadQueryAll` para rellenar la lista con los puntos de interés de la base de datos.

GameFragment

Este fragmente implementa el juego de preguntas sobre la Alhambra.

En primer lugar, al crear la vista del fragmento, se abre un diáologo definido en la función *explainDialog*, que muestra las intrucciones del juego. Al cerrar dicho diálogo, activamos la cuenta atrás , comenzamos a recibir datos del acelerómetro, y en el *listener* `mAccelerometerListener` comenzamos a esperar por uno de los dos gestos.

La cuenta atrás la gestiona `mCountDownTimer`. Este objeto tiene dos funciones, `onTick` que actualiza la barra de progreso cada 200ms, y `onFinish` que deshabilita el sensor para no recibir más respuestas, y muestra un mensaje informando de que el tiempo se ha acabado y los resultados.

Los dos gestos son mover el teléfono hacia adelante y atrás, y hacia los lados para el sí y el no respectivamente. Para reconocer estos gestos, comprobamos que el valor absoluto de la aceleración sea mayor a una determinada constante, respecto al eje Z para el si, y respecto al X para el no. Una vez reconocido alguno de los dos gestos, deja de escuchar para no mandar más de una respuesta involuntariamente, y se llama a la función `manageAnswer` con la respuesta correcta y la respuesta del usuario.

En primer lugar, la función `manageAnswer` comprueba si la respuesta es correcta o incorrecta comparando los dos booleanos que recibe, y muestra un mensaje acorde para informar al usuario. Tras esto, si todavía quedan preguntas, actualiza la pregunta actual, refresca la vista, y espera medio segundo. A continuación, vuelve a activar el `listener` para recibir la siguiente respuesta. En caso de no quedar preguntas disponibles, muestra el resultado final, y para la cuenta atrás.

Otras clases

InterestPoint

Esta clase corresponde a una simple abstracción de la información almacenada en un punto de interés.

```
data class InterestPoint(
        var id : Int,
        var title: String,
        var content : String
)
```

IPAdapter

Esta clase extiende a la clase `BaseAdapter`. La utilizamos en el fragmento correspondiente a la lista de puntos de interés `InterestPointMenuFragment`, donde mostramos una lista de todos los puntos de interés que se encuentran en la base de datos. El adaptador nos permite mostrar estos elementos en la lista de una forma más cómoda y personalizable. Podríamos haber utilizado un `ArrayAdapter` como se ha hecho en la lista de mapas interiores `InnerMapListFragment`, pero esto mermaba la escalabilidad de la aplicación a la hora de querer mostrar más información en la lista (por ejemplo, imágenes o un resumen del contenido).

DBManager

Esta clase nos permite utilizar la API de `SQLite` para almacenar los puntos de interés de nuestra aplicación. Dado que estos no cambian a lo largo de la ejecución podríamos haber utilizado un vector que los almacenara, pero vimos mas conveniente este modelo porque todos los fragmentos se pueden valer de la misma base de datos (por tener solo una actividad) y no tenemos que preocuparnos de tener un vector al que todos pudieran acceder.

La información almacenada en la base de datos es:

- Un identificador numérico (la clave primaria).
- El nombre del punto de interés.
- Toda la información al respecto del mismo.
- Su posición.
  - Se utiliza un valor numérico para indicar si el punto de interés se encuentra en el interior o dentro de algún edificio.
  - En el caso de ser un punto exterior, se utilizan latitud y longitud para determinarlo.
  - En caso de ser un punto interior, se utiliza un sistema de márgenes para posicionarlo en el mapa.

La clase dispone de 3 métodos importantes, todos ellos nos devuelven un cursor iterable sobre las filas resultantes en la base de datos.

- `queryById`. Nos permite obtener un punto de interés a partir de su ID.
- `queryByLocationType`. Nos permite obtener todos los puntos de interés que se encuentren en el exterior o dentro de un mismo edificio.
- `queryAll`. Nos permite obtener todos los puntos de interés.

GestureRecognitionDialog.

Esta es la clase que abre el diálogo al pulsar el botón para reconocer un punto de interés al hacer el gesto de apuntar, y que gestiona dicho gesto y su funcionalidad.

En primer lugar, para reconocer el gesto utilizamos dos sensores: el **acelerómetro** y el sensor de **campo magnético**. Para ello, definimos el *listener* `mPositionListener`. Este, una vez que posee datos no nulos de ambos sensores, comprueba que el móvil está en posición horizontal o más inclinado (como si apuntaras al suelo), lo que nos indica que el usuario ha realizado el gesto. Una vez reconocido el gesto, almacena el ángulo del móvil respecto al polo norte, lo que nos permitirá luego reconocer el punto de interés señalado. Tras esto, hace una petición de ubicación, y el *listener* de la ubicación será el encargado de identificar dicho punto de interés.

El *listener* encargado de obtener la ubicación actual del usuario es `mLocationListener`. Trataremos los valores de latitud y longitud como si fueran coordenadas sobre un plano. Dicho esto, una vez que conocemos la posición actual, calculamos el vector director de la semirrecta que comienza en nuestra localización y apunta en la dirección en la que apunta el móvil (utilizando el ángulo entre el eje X del teléfono y el polo norte almacenado anteriormente). Entonces, para reconocer a qué punto de interés estamos apuntando, descartamos los puntos que se quedan por detrás, calculamos la distancia a la semirrecta del resto y almacenamos el identificador del punto de interés que esté mas cerca de esta.

Por último, cerramos el diálogo y abrimos la vista asociada a la información de dicho punto de interés.