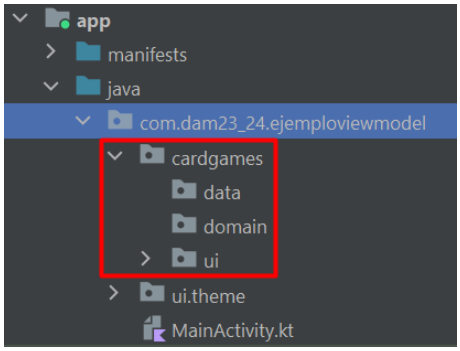
Aplicación con ViewModel y arquitectura MVVM.

El proyecto que se va a desarrollar en este ejemplo es una app que se llamará CardGames, que será una aplicación con un menú para acceder a dos posibles juegos de cartas: la carta más alta *(HighestCard)* y el blackjack *(BlackJack)*.

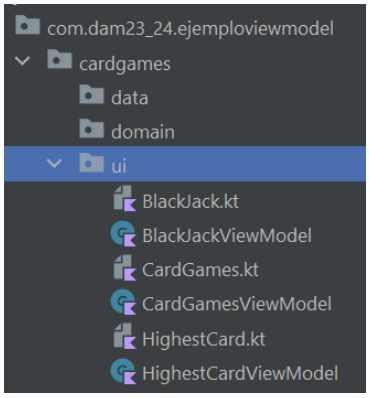
Os dejo un enlace, de lectura recomendable, dónde se explica de manera resumida cómo debe estar organizado un proyecto y en el que se ha basado esta práctica de ejemplo: [Android Studio Developers - Guía de arquitectura de apps]

Primero debemos preparar el proyecto y para eso vamos a crear una estructura de carpetas con el fin de organizar nuestro proyecto correctamente.

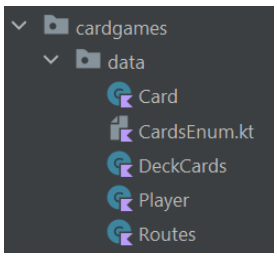
1. Creamos un paquete que contendrá nuestra única feature, `cardgames`, y dentro de ella, las carpetas básicas que nos van a servir para organizar nuestro código siguiendo unas buenas prácticas.



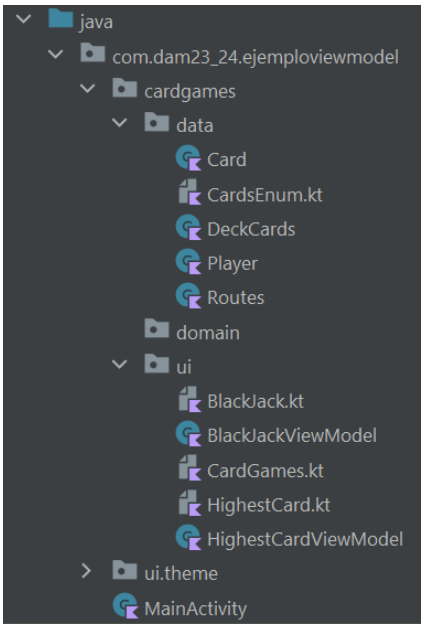
2. Creamos dentro de la carpeta ui las vistas o screeens *(**ficheros** de Kotlin que contendrán las funciones Composable con el layout)* y las **clases** View Model de cada una:



3. En el interior de la carpeta data, en este ejemplo sencillo, solo vamos a almacenar:
   - La clase *DeckCards* que va a gestionar una baraja de cartas
   - Las `data classes` *Player* y *Card* que almacenan la información de un jugador y una carta respectivamente
   - *CardsEnum:* `enum class` que representa los palos y cartas de la baraja.
   - Las rutas de navegación que hemos guardado en una `sealed class`.



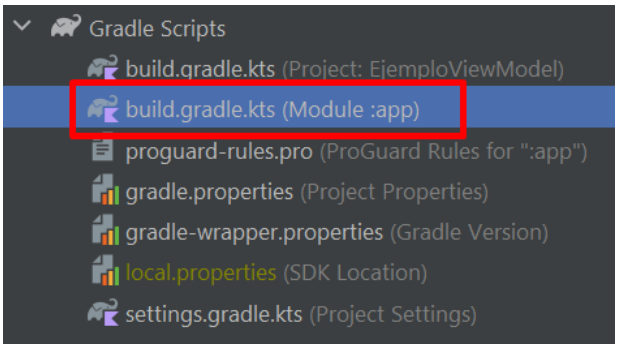
4. MainActivity lo vamos a dejar en la ruta por defecto dónde se genera inicialmente. Nuestro proyecto quedaría con la siguiente estructura de paquetes y ficheros:



A continuación, debemos insertar las dependencias necesarias para trabajar con la navegación y LiveData *(tendremos que sincronizar el gradle)*:

```
//Navigation
implementation("androidx.navigation:navigation-compose:2.7.5")

//LiveData
implementation("androidx.compose.runtime:runtime-livedata:1.5.4")
```




```
dependencies { this: DependencyHandlerScope

    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.lifecycle:lifecycle-runtime-ktx:2.6.2")
    implementation("androidx.activity:activity-compose:1.8.1")
    implementation(platform("androidx.compose:compose-bom:2023.03.00"))
    implementation("androidx.compose.ui:ui")
    implementation("androidx.compose.ui:ui-graphics")
    implementation("androidx.compose.ui:ui-tooling-preview")
    implementation("androidx.compose.material3:material3")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
    androidTestImplementation(platform("androidx.compose:compose-bom:2023.03.00"))
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    debugImplementation("androidx.compose.ui:ui-tooling")
    debugImplementation("androidx.compose.ui:ui-test-manifest")

    //Navigation
    implementation("androidx.navigation:navigation-compose:2.7.5")

    //LiveData
    implementation("androidx.compose.runtime:runtime-livedata:1.5.4")
}
```

Capa UI

Cada vista debe tener su screen y viewmodel que va a gestionar los cambios de estados que obligaran a recomponerse a los distintos componentes gráficos de la screen.

La nomenclatura de los ficheros que contendrán las clases ViewModel será el mismo nombre de la Screen, pero agregando el sufijo ViewModel.

En nuestro ejemplo vamos a tener 3 pantallas: CardGames, HighestCard y BlackJack.

- **CardGames** será un menú simple para movernos a los juegos de la carta más alta (HighestCard) y el blackjack (BlackJack). Tendrá navegación, pero no va a necesitar gestionar ningún estado, por lo que no vamos a crear un ViewModel para esta Screen.
- **HighestCard** mostrará una baraja de cartas, ya barajada, y dos botones: uno en el que podrá pedir una carta y otro que volverá a introducir todas las cartas de la baraja y barajar de nuevo. En esta pantalla se va a controlar que al volver a CardGames se reinicie la baraja.
- **BlackJack** es el juego de blackjack en modo 2 jugadores. Primero mostrará un diálogo que solicitará el apodo o nickname de ambos jugadores y al pulsar el botón Aceptar creará una partida nueva. Ambos jugadores pueden pedir una Carta o plantarse para tener una puntuación de 21 o lo más cercana sin pasarse.

A la hora de crear las clases de ViewModel seguiremos los siguientes pasos:

1. La clase será heredada de ViewModel:

```
class HighestCardViewModel : ViewModel() {
    …
}
```

   El problema es que nuestro ViewModel va a tener que ejecutar un método de DeckCards que necesita el contexto. Para solucionar esto, vamos a crear el ViewModel heredando de una subclase de ViewModel, llamada AndroidViewModel,

que contiene el contexto de la aplicación donde se inicializó el ViewModel. De esta manera podremos tener el contexto en una variable privada dentro del ViewModel:

```
Diego Cano Sibón
class HighestCardViewModel(application: Application) : AndroidViewModel(application) {
// If we don't need to use the context inside, it's better to inherit from ViewModel
//class HighestCardViewModel : ViewModel() {

    @SuppressLint("StaticFieldLeak")
    private val context = getApplication<Application>().applicationContext

    private val _imageId = MutableLiveData<Int>()
    val imageId : LiveData<Int> = _imageId

    private val _imageDesc = MutableLiveData<String>()
    val imageDesc : LiveData<String> = _imageDesc

    private val _card = MutableLiveData<Card>()

    /**
     * Initializes the ViewModel by restarting the game.
     */
    Diego Cano Sibón
    init {
        restart()
    }
```

Los estados que antes declarábamos dentro de nuestras funciones Composable, ahora estarán contenidos en el ViewModel, cómo podéis observar en la imagen anterior.

Para cada estado declararemos una variable privada y mutable de tipo ***MutableLiveData*** con el tipo de dato de nuestro estado *(Int, String, Boolean, Card, Player, etc.)* que solo gestionaremos dentro del ViewModel y una variable pública inmutable de tipo ***LiveData***, que será la que utilizaremos en el fichero Screen para escribir el valor de nuestro estado.

La clase ViewModel también tendrá los métodos necesarios para actualizar y gestionar los estados.

```
/**
 * Gets a new card from the deck and updates LiveData values accordingly.
 */
Diego Cano Sibón
fun getCard() {
    _card.value = DeckCards.getCard()
    _imageId.value = _card.value?.idDrawable
    _imageDesc.value = "${_card.value?.name} de ${_card.value?.suit}"
}

/**
 * Restarts the deck, shuffles it, and sets the initial face-down card.
 */
Diego Cano Sibón
fun restart() {
    DeckCards.newDeckOfCards(context)
    DeckCards.shuffle()
    _card.value = DeckCards.getFaceDownCard()
    _imageId.value = _card.value?.idDrawable
    _imageDesc.value = ""
}
```

En el fichero screen, que contiene solo la parte visual, es decir, las funciones Composable, debemos pasar cómo argumento el objeto ViewModel. Para ello, haremos algo similar a lo que ya hicimos en la navegación:

```
/**
 * Composable function representing the screen for the Highest Card game.
 *
 * @param navController The navigation controller used for navigating to different screens.
 * @param highestCardViewModel The ViewModel responsible for managing the Highest Card game logic.
 */
Diego Cano Sibón
@Composable
fun HighestCardScreen(
    navController: NavHostController,
    highestCardViewModel: HighestCardViewModel
) {
    val imagenId: Int by highestCardViewModel.imageId.observeAsState(initial = 0)
    val descImagen: String by highestCardViewModel.imageDesc.observeAsState(initial = "")
```

Dentro de la función Composable, vamos a declarar una variable que hará referencia a la variable pública del ViewModel. Esta variable estará “observando” si se produce algún cambio para actualizarse y de esta forma recomponer la vista dónde sea necesario por ese cambio del estado.

```
val imagenId: Int by highestCardViewModel.imageId.observeAsState(initial = 0)
```

Como norma básica, la función adecuada para declarar las variables será una función que me permita elevar el estado con los mínimos pasos de parámetros posibles.

Es decir, intentar declarar la variable en la función Composable llamadora justo superior o anterior a la función dónde se utilizará el estado cuando sea posible.

Si el estado se utiliza en varias funciones separadas, puede que sea necesario elevar el estado a una función superior, desde dónde se pueda pasar por argumento hasta llegar a dichas funciones.

Dónde se crea el objeto ViewModel y se pasa al Screen

El lugar adecuado para crear los objetos ViewModel y pasarlos a las Screens será el MainActivity. Dentro de esta clase, declararemos cada objeto de tipo NombreScreenViewModel de la siguiente manera:

```
Diego Cano Sibón
class MainActivity : ComponentActivity() {

    private val highestCardViewModel: HighestCardViewModel by viewModels()
    private val blackJackViewModel: BlackJackViewModel by viewModels()
```

Para posteriormente pasarlos cómo argumentos en las llamadas de las funciones Composable principales que representan cada Screen:

```
composable(HighestCardScreen.route) { this: AnimatedContentScope  it: NavBackStackEntry
    HighestCardScreen(
        navController = navController,
        highestCardViewModel = highestCardViewModel
    )
}
composable(BlackJackScreen.route) { this: AnimatedContentScope  it: NavBackStackEntry
    BlackJackScreen(
        navController = navController,
        blackJackViewModel = blackJackViewModel
    )
```