

# Trabajo de Programación Multimedia y Dispositivos Móviles.

## Blackjack: Un paseo por la programación funcional y el modelo MVVM.

2º de Desarrollo de Aplicaciones Multiplataforma



*Por Agustín Rodríguez Márquez*

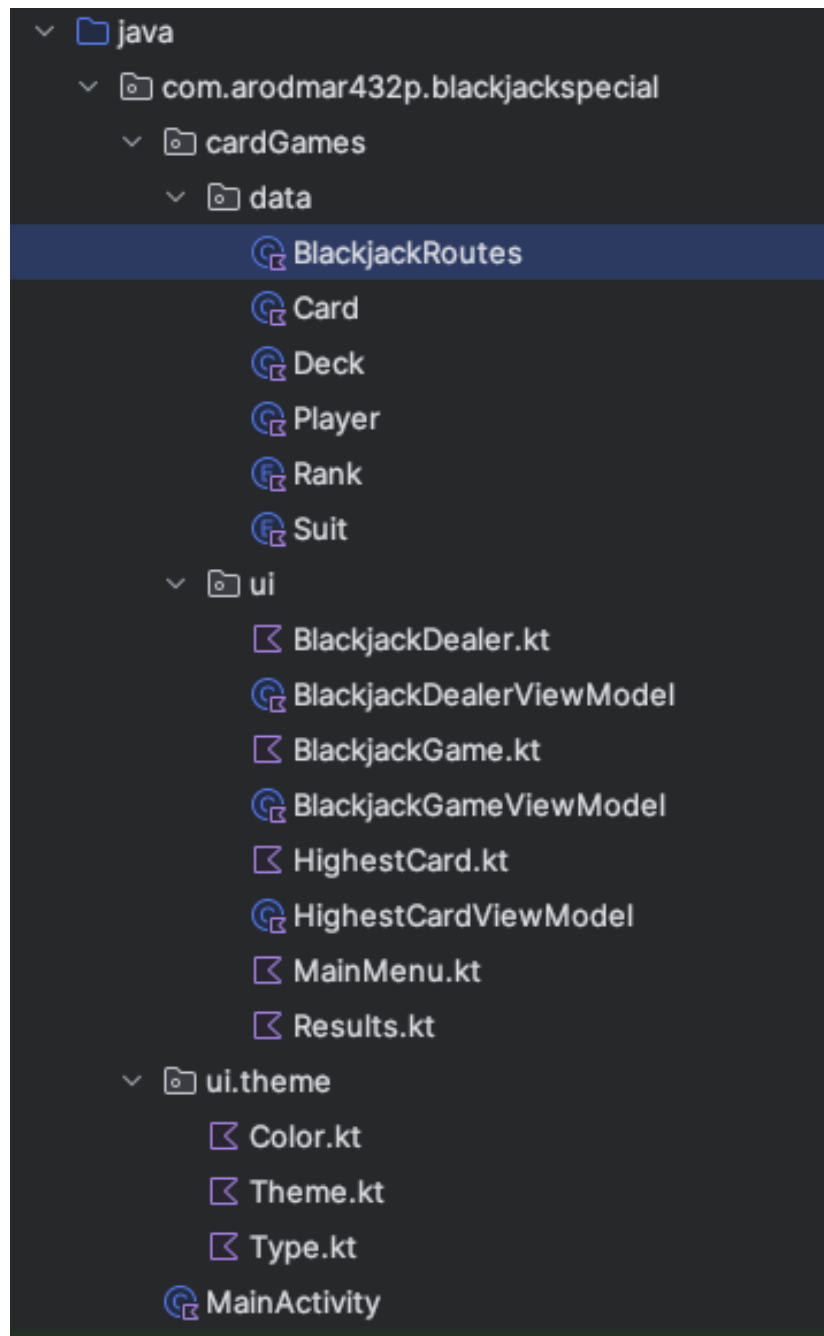
*IES Rafael Alberti*

## INDICE

Introducción.....	3
Modelo.....	4
BlackjackRoutes.....	4
Card.....	4
Suit.....	5
Rank.....	5
Player.....	6
Deck.....	6
ui, vista-Modelo y Vista.....	9
BlackjackDealerViewModel.....	9
BlackjackDealerScreen.....	12
BlackjackGameViewModel.....	15
BlackjackGame.....	20
HighestCardViewModel.....	23
HighestCard.....	24
MainMenu.....	26
Results.....	27
MainActivity.....	28
Conclusiones.....	29

## Introducción

Para realizar este trabajo de Blackjack he usado una estructura Modelo Vista Vista-Modelo. La estructura del proyecto viene determinada por los requisitos del ejercicio.



Como se puede ver en la imagen anterior, tenemos un paquete cardGames, idéntico a la estructura que se nos proporcionó durante las clases, a partir de ahí, he creado un paquete data para las clases del modelo y un paquete ui para las clases que gestionan la lógica de negocio de la aplicación y los archivos kt correspondientes a la vista.

## El modelo:

Las clases que conforman el modelo son bastante básicas y viniendo del trabajo de la carta más alta, apenas han tenido modificación desde entonces.

### BlackjackRoutes

Una de las clases de nueva creación ha sido la clase BlackjackRoutes, he creado esta clase para usar NavController tal como vimos en clase. Aquí se definen los objetos que serán las screens a las que navegaremos.

```
/**
 * A sealed class representing the different routes in the Blackjack
 game.
 *
 * @property route The string representation of the route.
 */
sealed class BlackjackRoutes(val route: String) {

    /**
     * Represents the main menu screen route.
     */
    object MainMenuScreen : BlackjackRoutes("MainMenuScreen")

    /**
     * Represents the Blackjack game screen route.
     */
    object BlackjackScreen : BlackjackRoutes("BlackjackScreen")

    /**
     * Represents the Blackjack dealer screen route.
     */
    object BlackjackDealerScreen :
        BlackjackRoutes("BlackjackDealerScreen")

    /**
     * Represents the results screen route.
     */
    object ResultsScreen : BlackjackRoutes("ResultsScreen")

    /**
     * Represents the Highest Card Game Screen
     */
    object HighestCardScreen : BlackjackRoutes("HighestCardScreen")
}
```

### Card

La clase Card representa a una carta, estará compuesta por el valor de la carta y el palo de la misma. Además, le he añadido las variables minPoints y maxPoints que servirán para el cálculo de puntos.

idDrawable lo he marcado con @DrawableRes como en el proyecto del profesor, porque a pesar de que en principio no lo declaré de esta forma, sino que lo declaré como String y funcionaba correctamente, esta anotación proporciona verificaciones de tiempo de compilación para los drawable. He añadido el símbolo de interrogación para el escenario

en el que no tenga una imagen asociada se permita que tenga un valor nullable, lo he considerado útil durante todo el proyecto, especialmente en las primeras fases cuando podía manejar casos y las imágenes pudiesen no estar disponibles.

```
/**
 * A data class representing a card in the game of Blackjack.
 */
* @property rank The rank of the card.
* @property suit The suit of the card.
* @property minPoints The minimum points the card can contribute in the game.
* @property maxPoints The maximum points the card can contribute in the game.
* @property idDrawable The drawable resource ID of the card image, if available.
*/
@Agustrodmar
data class Card(
    val rank: Rank,
    val suit: Suit,
    val minPoints: Int,
    val maxPoints: Int,
    @DrawableRes val idDrawable: Int?
)
```

## Suit

La clase Suit es simplemente una enum class con los palos de la baraja francesa.

```
enum class Suit {
    HEARTS,
    DIAMONDS,
    CLUBS,
    SPADES
}
```

## Rank

La clase rank representa el número de las cartas en el juego, que van del As al Rey.

```
enum class Rank {
    ACE,
    TWO,
    THREE,
    FOUR,
    FIVE,
    SIX,
    SEVEN,
    EIGHT,
    NINE,
    TEN,
    JACK,
    QUEEN,
    KING
}
```

## Player

La clase Player, como la clase Card, es simplemente una clase data, aquí se guarda el nombre, la mano (como una lista mutable de las cartas que el jugador tendrá en su mano), y por otro lado los puntos del jugador.

```
data class Player(  
    val name: String,  
    var hand: MutableList<Card> = mutableListOf(),  
    var points: Int  
)
```

## Deck

De todo el modelo, la clase deck es aquella que representa la baraja, y es la que reviste más complejidad de todas ellas.

La función createDeck es una función estática en el companion object. La función va a aceptar un mapa de imágenes de cartas como parámetro, que mapea los nombres de las cartas a sus correspondientes recursos de imagen.

A continuación, crea una lista vacía de cartas, luego recorre cada palo y cada rango de cartas con un doble bucle for. Para cada combinación de palo y rango, calcula los puntos mínimos y máximos de la carta. Los ases pueden valer 1 o 11 puntos por la lógica del blackjack, las figuras valen 10 puntos y el resto de cartas valen su valor nominal.

Posteriormente, construye el nombre del recurso de imagen para la carta concatenando el nombre del palo y el rango. Este nombre se utiliza para buscar el ID del recurso de imagen en el mapa de imágenes de cartas.

Si el ID del recurso de imagen no es null, se crea una nueva carta con el rango, el palo, los puntos mínimos y máximos, y el ID del recurso de imagen, y se añade a la lista de cartas.

```
/**  
 * A class representing a deck of cards in the game of Blackjack.  
 */  
@property cardImageMap A map containing the image resources for the  
cards.  
*/  
class Deck(private val cardImageMap: Map<String, Int>) {  
    /**  
     * Companion object to create a deck of cards.  
     */  
    companion object {  
        /**  
         * Creates a deck of cards.  
         *  
         * @param cardImageMap A map containing the image resources  
         for the cards.  
         * @return An ArrayList of Card objects representing a deck of  
         cards.  
         */  
        fun createDeck(cardImageMap: Map<String, Int>):
```

```

ArrayList<Card> {
    val cardsList = ArrayList<Card>()
    // Loop through each suit
    for (suit in Suit.values()) {
        // Loop through each rank
        for (rank in Rank.values()) {
            // Determine the minimum and maximum points for
the card
            val minPoints =
                if (rank == Rank.ACE) 1 else if (rank.ordinal
> 10) 10 else rank.ordinal
            val maxPoints =
                if (rank == Rank.ACE) 11 else if (rank.ordinal
> 10) 10 else rank.ordinal

            // Construct the drawable name for the card image
            val idDrawableName = when (suit) {
                Suit.HEARTS -> "corazones"
                Suit.DIAMONDS -> "diamantes"
                Suit.CLUBS -> "treboles"
                Suit.SPADES -> "picas"
            } + when (rank) {
                Rank.ACE -> "a"
                Rank.TWO -> "2"
                Rank.THREE -> "3"
                Rank.FOUR -> "4"
                Rank.FIVE -> "5"
                Rank.SIX -> "6"
                Rank.SEVEN -> "7"
                Rank.EIGHT -> "8"
                Rank.NINE -> "9"
                Rank.TEN -> "10"
                Rank.JACK -> "j"
                Rank.QUEEN -> "q"
                Rank.KING -> "k"
            }

            // Get the drawable ID from the map
            val idDrawable = cardImageMap[idDrawableName]
            // If the drawable ID is not null, add the card to
the list
            if (idDrawable != null) {
                cardsList.add(Card(rank, suit, minPoints,
maxPoints, idDrawable))
            }
        }
    }
    // Return the list of cards
    return cardsList
}

// Create the deck of cards
val cardsList = createDeck(cardImageMap)

/**
 * Resets the deck of cards.
 *
 * This function recreates the deck to ensure all cards are

```

```

present.
    */
    private fun reset() {
        cardsList.clear()
        cardsList.addAll(createDeck(cardImageMap))
    }

```

El resto de funciones que aparecen en la clase Deck, como shuffle(), son llamadas al método shuffle para la lista de cartas más una llamada a la función reset(), que se encarga de volver a añadir las cartas que se han ido usando, lo que baraja el mazo al completo. La función hasCards(), que indica si a la baraja le quedan cartas, o getCard(), que gestiona una excepción en caso de que la baraja esté vacía, llamando a shuffle al final de cada ronda no veo posible que se agote la baraja, de ahí que haya creado una excepción para evitar un cuelgue en este caso.

```

/**
 * Shuffles the deck of cards.
 *
 * This function first resets the deck to ensure all cards are
 * present,
 * then shuffles the cards.
 */
fun shuffle() {
    reset()
    cardsList.shuffle()
}

/**
 * Checks if the deck has cards.
 *
 * @return A Boolean indicating whether the deck has cards.
 */
fun hasCards(): Boolean {
    return cardsList.isNotEmpty()
}

/**
 * Gets a card from the deck.
 *
 * @return A Card object.
 * @throws IllegalStateException If the deck is empty.
 */
fun getCard(): Card {
    if (cardsList.isEmpty()) {
        throw IllegalStateException("No se puede obtener una carta
porque el mazo está vacío.")
    }
    return cardsList.removeAt(cardsList.size - 1)
}
}

```



## ui, Vista-Modelo y Vista

Dentro de la carpeta ui, se encuentran las dos clases de vista-Modelo, en adelante viewModel, que he usado para el programa. Como el programa tiene un modo contra la máquina y un modo contra otro jugador, he creado dos viewModel distintos para cada modo de juego, cada uno usa su lógica de negocio de distinta forma.

### BlackjackDealerViewModel

Primero repasaré BlackjackDealerViewModel. Lo primero de todo ha sido la declaración de las variables que usaré. La mayoría, como las pasaré a la vista, son MutableLiveData. Jugador y Player son instancias de la clase jugador, con sus propios nombres, una mutable list que es la mano del jugador, y la puntuación del mismo. También he creado una variable de dealSoundPlayer del tipo MediaPlayer inicializada como null, para introducir la música en el juego. Lo más interesante es el objeto deck, que contiene un mapa de imágenes con los drawable de las cartas. Este ha sido sin duda el punto de inflexión en la forma en la que manejaba el programa, pues estuve a punto de añadir los drawable en la clase Deck. Decidí añadir los drawable aquí y no en la clase deck para aprovechar las ventajas que me podía dar el usar un companion object. La desventaja es que creo un objeto deck con el mapa de imágenes en cada uno de los viewModel, por el contrario, y lo que considero un gran beneficio, es que, al hacerlo así, puedo incluir drawables diferentes dependiendo del modo de juego.

```
private val deck = Deck(cardImageMap = mapOf("corazones" to
R.drawable.corazones,
    "corazones2" to R.drawable.corazones2,
    ...
))

// The player
private val player = Player("Player", mutableListOf(), 0)

// The dealer
private val dealer = Player("La mesa", mutableListOf(), 0)

// LiveData objects to hold the game state
val winner = MutableLiveData<String>()
val playerPoints = MutableLiveData<Int>()
private val dealerPoints = MutableLiveData<Int>()
val playerHand = MutableLiveData<List<Card>>()
val dealerHand = MutableLiveData<List<Card>>()
val gameInProgress = MutableLiveData<Boolean>()
val isGameOver = MutableLiveData(false)
val gameReset = MutableLiveData(false)

// MediaPlayer to play the shuffle sound
private var dealSoundPlayer: MediaPlayer? = null
```

En BlackjackDealerViewModel se gestionan las funciones startGame(), que baraja el mazo y limpia la mano de los jugadores, además de añadir dos cartas de inicio a cada jugador llamando a getCard(), de la clase Deck. Luego se llama a la función calculatePoints(), que se encarga de sumar el valor en las cartas de los jugadores.

La función dealerTurn() tiene una lógica sencilla. Básicamente, mientras que el dealer siga teniendo menos de 17 puntos, seguirá pidiendo cartas y lo más importante, llamar

a `endTurn()` para que su turno termine de forma automática después de esto, de lo contrario no se cedería el turno al jugador.

Por su parte, `calculateCardPoints()`, incluye la lógica necesaria para que cambie el valor del as dependiendo de cómo sea necesario para que sume correctamente, es decir, si tengo dos ases en la mano, tendré un total de 12 puntos y no 22, con vistas a no pasarme, de lo contrario, se llama a sumar los `maxPoints` de la clase `Card`.

```
private fun dealerTurn() {
    // While the dealer has less than 17 points and the deck has
    cards, add a card to the dealer's hand
    while (dealer.points < 17 && deck.hasCards()) {
        dealer.hand.add(deck.getCard())
        dealerHand.value = dealer.hand
        calculatePoints()
    }
    endTurn()
}

/**
 * Ends the player's turn and starts the dealer's turn.
 */
fun stand() {
    dealerTurn()
}

/**
 * Calculates the points for a player.
 *
 * @param player The player to calculate points for.
 * @return The total points.
 */
private fun calculateCardPoints(player: Player): Int {
    var total = 0
    var aces = 0

    // Calculate the total points, taking into account the value of
    aces
    for (card in player.hand) {
        total += if (card.rank != Rank.ACE) {
            if (card.rank.ordinal >= Rank.JACK.ordinal) 10 else
card.rank.ordinal + 1
        } else {
            aces++
            card.maxPoints
        }
    }

    // If the total is over 21 and there are aces, subtract 10 for
    each ace
    while (total > 21 && aces > 0) {
        total -= 10
        aces--
    }

    return total
}
```

calculatePoints() y endTurn() ya han sido mencionadas con anterioridad, calculatePoints() gestiona los puntos de la mano de cada jugador, endTurn() se compone de un when que gestiona los sucesos que pueden darse para que haya un ganador, como que el jugador, o el Dealer, tengan más puntos el uno que el otro, o que haya empate. Por último se llama a la booleana isGameOver() y se establece en true. Cuando isGameOver es true, abro un dialogo en la vista. Cuando el usuario pulsa “Aceptar” sobre el Dialog, isGameOver, se vuelve a establecer en false y establezco en true gameReset.

Para la música creo una función playDealSound(), con un parámetro context que luego pasaré a la vista. Lo he envuelto en un try catch porque si por algún motivo el usuario no puede activar el sonido, se evita el cuelgue de la aplicación.

```
private fun calculatePoints() {
    for (currentPlayer in listOf(player, dealer)) {
        currentPlayer.points = calculateCardPoints(currentPlayer)
        if (currentPlayer == player) {
            playerPoints.value = currentPlayer.points
        } else {
            dealerPoints.value = currentPlayer.points
        }
    }
}

/**
 * Ends the turn and determines the winner.
 */
private fun endTurn() {
    when {
        player.points > 21 -> winner.value = "Dealer"
        dealer.points > 21 -> winner.value = "Player"
        player.points > dealer.points -> winner.value = "Player"
        dealer.points > player.points -> winner.value = "Dealer"
        else -> winner.value = "Draw"
    }
    isGameOver.value = true
}

/**
 * Closes the game over dialog and resets the game.
 */
fun closeDialog() {
    isGameOver.value = false
    gameReset.value = true
}

/**
 * Plays the deal sound.
 *
 * @param context The context to use to create the MediaPlayer.
 */
fun playDealSound(context: Context) {
    try {
        dealSoundPlayer = MediaPlayer.create(context,
R.raw.repartir)
        dealSoundPlayer?.start()
    } catch (e: Exception) {
        Log.e("BlackjackDealerViewModel", "Error playing deal
sound: ${e.message}")
    }
}
```

```
}  
}  
}
```

## BlackjackDealerScreen

Esta la función componible principal de la vista del modo contra la máquina, como se puede ver, llamo a los LiveData con los observeAsState.

He vaciado la vista de cualquier tipo de lógica de negocio. En la siguiente imagen puede verse el Dialog que mencioné con anterioridad. En este Dialog, llamo a la función closeDialog(), que se encargará de resetear la partida.

```
@Composable  
fun BlackjackDealerScreen(blackjackDealerViewModel:  
BlackjackDealerViewModel) {  
    // Get the game state from the ViewModel  
    val playerPoints by  
blackjackDealerViewModel.playerPoints.observeAsState(0)  
    val winner by blackjackDealerViewModel.winner.observeAsState("")  
    val playerHand by  
blackjackDealerViewModel.playerHand.observeAsState(listOf())  
    val dealerHand by  
blackjackDealerViewModel.dealerHand.observeAsState(listOf())  
    val gameInProgress by  
blackjackDealerViewModel.gameInProgress.observeAsState(false)  
    val isGameOver by  
blackjackDealerViewModel.isGameOver.observeAsState(false)  
  
    // Display the dealer screen  
    Box(modifier = Modifier.fillMaxSize()) {  
        Image(  
            painter = painterResource(id = R.drawable.tapete),  
            contentDescription = null,  
            modifier = Modifier.fillMaxSize(),  
            contentScale = ContentScale.FillBounds  
        )  
  
        // Display the game screen or the start screen  
        if (gameInProgress || isGameOver) {  
            GameScreen(blackjackDealerViewModel, playerPoints,  
playerHand, dealerHand)  
        } else {  
            StartScreen(blackjackDealerViewModel)  
        }  
  
        // Display a dialog when the game is over  
        if (isGameOver) {  
            AlertDialog(  
                onDismissRequest = {  
blackjackDealerViewModel.closeDialog() },  
                title = { Text(text = "Fin de la ronda") },  
                text = {  
                    if (winner == "Draw") {  
                        Text(text = "Empate")  
                    } else {  
                        Text(text = "El ganador es: $winner")  
                    }  
                }  
            ),  
        },  
    },  
}
```

```

        confirmButton = {
            val context = LocalContext.current
            Button(onClick = {
                blackjackDealerViewModel.closeDialog()
                blackjackDealerViewModel.startGame()

                blackjackDealerViewModel.playDealSound(context)
            }, colors =
                ButtonDefaults.buttonColors(containerColor = Color(0xFFD4AF37)),
                border = BorderStroke(2.dp, Color.White)
            ) {
                Text("Aceptar")
            }
        }
    }
}

```

StartScreen() es solo una sencilla componible que contiene un botón al inicio de cada partida:

```

/**
 * A composable function to display the start screen of the Blackjack
 * game.
 *
 * @param blackjackDealerViewModel The ViewModel for the dealer.
 * @param winner The winner of the game.
 */
@Composable
fun StartScreen(blackjackDealerViewModel: BlackjackDealerViewModel) {

    // Get the game reset state from the ViewModel
    val gameReset by
        blackjackDealerViewModel.gameReset.observeAsState(false)

    // Start a new game when the game is reset
    if (gameReset) {
        blackjackDealerViewModel.startGame()
        blackjackDealerViewModel.gameReset.value = false
    }

    // Display the start screen
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceEvenly
    ) {

        // Start game button
        Button(
            onClick = { blackjackDealerViewModel.startGame() },
            colors = ButtonDefaults.buttonColors(containerColor =
                Color(0xFFD4AF37)),
            border = BorderStroke(2.dp, Color.White)
        ) {
            Text("Empezar Partida", color = Color.Black)
        }
    }
}

```

```

    }
}

```

GameScreen(), por su parte, es la componible más extensa del archivo, esto es debido a que es la función que almacena más botones, además de disposiciones como la de las cartas, para las que he usado “.offset” con el fin de que apareciesen en diagonal. También se muestran los puntos del jugador.

Para dealerHand() me he preocupado de asegurarme de que sus cartas, excepto la primera (index 0), y solo si el juego no está acabado, se muestren con el drawable bocabajo, en cualquier otra circunstancia se dibujan con su idDrawable respectivo. Con esto consigo que las cartas de la mesa no se enseñen, y la apuesta tenga un mayor sentido.

```

@Composable
fun GameScreen(
    blackjackDealerViewModel: BlackjackDealerViewModel,
    playerPoints: Int,
    playerHand: List<Card>,
    dealerHand: List<Card>
) {
    // Get the game over state from the ViewModel
    val isGameOver by
blackjackDealerViewModel.isGameOver.observeAsState(false)

    // Display the game screen
    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceEvenly
    ) {

        // Display the dealer's cards
        Row(
            horizontalArrangement = Arrangement.spacedBy((-80).dp)
        ) {
            dealerHand.forEachIndexed { index, card ->
                val cardResource = if (index != 0 && !isGameOver)
R.drawable.bocabajo else card.idDrawable
                Box(
                    modifier = Modifier
                        .size(90.dp, 150.dp)
                        .offset { IntOffset((index *
50).dp.roundToPx(), (index * 20).dp.roundToPx()) }
                ) {
                    Image(
                        painter = painterResource(id =
cardResource!!),
                        contentDescription = "Dealer Card"
                    )
                }
            }
        }

        Text(text = "Player Points: $playerPoints", color =
Color.White)

        Row(

```

```

        horizontalArrangement = Arrangement.spacedBy((-80).dp)
    ) {
        playerHand.forEachIndexed { index, card ->
            Box(
                modifier = Modifier
                    .size(90.dp, 150.dp)
                    .offset { IntOffset((index *
50).dp.roundToPx(), (index * 20).dp.roundToPx()) }
            )
        }
    }

```

## BlackjackGameViewModel

Este fue el primer viewmodel sobre el que trabajé en el proyecto, su lógica de funcionamiento es algo distinta a la del anterior viewModel.

Lo primero que hago es crear el objeto de la clase Deck con el mapa de drawables para este modo de juego. Posteriormente creo los LiveData que luego le pasaré a la Vista.

```

/**
 * A ViewModel class representing the game of Blackjack.
 */
class BlackjackGameViewModel : ViewModel() {
    // The deck of cards
    private val deck = Deck(cardImageMap = mapOf("corazones" to
R.drawable.corazones,
        "corazones2" to R.drawable.corazones2,
        "corazones3" to R.drawable.corazones3,

    ))

    // LiveData objects to hold the game state
    private val _players = MutableLiveData<List<Player>>(emptyList())
    val players: LiveData<List<Player>> get() = _players
    private val _winner = MutableLiveData<Player?>()
    val winner: LiveData<Player?> get() = _winner
    private val _currentTurn = MutableLiveData<Player>()
    val currentTurn: LiveData<Player> get() = _currentTurn
    private val _gameInProgress = MutableLiveData<Boolean>()
    val gameInProgress: LiveData<Boolean> get() = _gameInProgress

    private val _showDialog = MutableLiveData<Boolean>() // To declare
the winner
    val showDialog: LiveData<Boolean> get() = _showDialog // To handle
my Results screen

    private val _player1Wins = MutableLiveData(0)
    val player1Wins: LiveData<Int> get() = _player1Wins

    private val _player2Wins = MutableLiveData(0)
    val player2Wins: LiveData<Int> get() = _player2Wins

    private val _eventCloseApp = MutableLiveData<Boolean>()
    val eventCloseApp: LiveData<Boolean> get() = _eventCloseApp

    private var mediaPlayer: MediaPlayer? = null

    private var isPlaying = false

```

Para este modo de juego uso dos contadores que guardarán los resultados y se los llevará a la pantalla de Resultados.

En el init, inicio las variables que voy a usar. La gestión del Dialog es algo distinta aquí, aunque sigo usando un booleano para controlar su aparición.

En la fun endGame(), se puede ver como he establecido la lógica de los contadores para que sumen +1 en caso de victoria de un jugador u otro. En cualquiera de los casos, \_showDialog se establece en true.

```
init {
    _gameInProgress.value = false
    val player1 = Player("Player 1", mutableListOf(), 0)
    val player2 = Player("Player 2", mutableListOf(), 0)
    _players.value = listOf(player1, player2)
    _currentTurn.value = player1
    checkForBlackjack()
    _showDialog.value = false
}

/**
 * Starts a new game.
 */
fun startGame() {
    _winner.value = null
    _showDialog.value = false
    restartGame()
    deck.shuffle()
    startDeal()
    _gameInProgress.value = true
}

/**
 * Ends the game and declares the winner.
 * @param winner The winner of the game.
 */
private fun endGame(winner: Player?) {
    if (winner != null) {
        _winner.value = winner
        _showDialog.value = true
        if (winner.name == "Player 1") {
            _player1Wins.value = (_player1Wins.value ?: 0) + 1
        } else if (winner.name == "Player 2") {
            _player2Wins.value = (_player2Wins.value ?: 0) + 1
        }
    } else {
        _winner.value = null
        _showDialog.value = true
    }
}
```

closeDialog() funciona ligeramente distinto en este modo. Cuando el usuario pulsa en Aceptar sobre el Dialog, \_showDialog se establece en false, pero aquí \_gameInProgress sigue false ya que al final de cada partida, la partida no se inicia automáticamente, sino que devuelve a la pantalla previa. Hice esto con intención de añadir más lógica (como apuestas) entre partida y partida. Aunque finalmente descarté la idea de las apuestas, no he visto necesidad de cambiar esta pantalla.



startDeal() reparte carta a los jugadores en un rango a 2. A diferencia del anterior viewModel, donde en startGame() llamaba a player.hand.add una vez por cada carta, en este viewModel creé una función específica para el primer reparto, así como para los repartos consecutivos, representados en la fun hitMe().

Lo más interesante en hitMe(), aparte de llamar a requestCard() para solicitar una carta, quizás sea que llamo a passTurn() automáticamente una vez el jugador pide carta. Me parecía interesante que solo se pudiese pedir una carta, al ser 1vs1 un modo de juego inventado, y al no tener experiencia previa con el juego de Blackjack, esto me pareció la lógica correcta del juego para hacerlo más divertido, de modo que, al pedir la tercera carta, los jugadores comparan su puntuación y se decide quien gana.

```
/**
 * Deals a specified number of cards to each player.
 * @param numCards The number of cards to deal.
 */
private fun startDeal() {
    for (i in 0 until 2) {
        for (player in _players.value!!) {
            player.hand.add(deck.getCard())
        }
    }
}

/**
 * Handles the player's turn when they choose to hit.
 *
 * @param player The player who is hitting.
 */
fun hitMe(player: Player) {
    if (_currentTurn.value == player) {
        requestCard(player)
        if (_gameInProgress.value == true) {
            passTurn()
        }
    }
}

/**
 * Handles the player's turn when they choose to pass.
 *
 * @param player The player who is passing.
 */
fun pass(player: Player) {
    if (_currentTurn.value == player) {
        passTurn()
    }
}
```

requestCard() está estrechamente relacionada con hitMe(), pero requestCard se encarga de la lógica de añadir una carta a la mano del jugador desde la baraja. Después de añadir la carta verifica que si el jugador se pasa de 21.

La fun pass(), que es a la que llama el botón de la vista, se encarga de llamar a passTurn(). Una lógica muy simple para cuando el usuario decida pasar turno.

passTurn() se encarga de pasar el turno al otro jugador. Algo más compleja que las anteriores, passTurn identifica al jugador que tiene el turno y cambia el valor de \_currentTurn. Si el siguiente jugador vuelve a ser el primero, quiere decir que ya ambos han hecho su apuesta, por lo que se comparan las puntuaciones y se determina el ganador. Haciendo una analogía, diría que passTurn() se encarga de la lógica de bajo nivel detrás de pass().

```
/**
 * Requests a card for a player.
 *
 * @param player The player who is requesting a card.
 */
private fun requestCard(player: Player) {
    player.hand.add(deck.getCard())

    if (calculatePoints(player.hand) > 21) {
        endGame(_players.value?.first { it != player }!!)
    }
}

/**
 * Passes the turn to the next player.
 */
private fun passTurn() {
    val currentPlayer = _currentTurn.value
    val nextPlayer = _players.value?.let { players ->
        val currentIndex = players.indexOf(currentPlayer)
        val nextIndex = (currentIndex + 1) % players.size
        players[nextIndex]
    }
    _currentTurn.value = nextPlayer

    if (_currentTurn.value == _players.value?.first()) {
        val currentPlayerPoints =
            calculatePoints(currentPlayer!!.hand)
        val nextPlayerPoints = calculatePoints(nextPlayer!!.hand)
        val winner = when {
            currentPlayerPoints > 21 -> nextPlayer
            nextPlayerPoints > 21 -> currentPlayer
            currentPlayerPoints > nextPlayerPoints -> currentPlayer
            currentPlayerPoints < nextPlayerPoints -> nextPlayer
            else -> null // // This is how I plan to handle a tie
        }
        endGame(winner)
    }
}
```

Siguiendo una lógica similar al anterior viewModel, tengo calculateCardPoints(), que sencillamente calcula el valor de las cartas, por lo que me aseguro de que el As, dependiendo de la circunstancia, para adoptar el valor 1 u 11, y que las figuras todas valgan 10.

calculatePoints(), por otro lado, calcula el total de puntos de la mano de cartas del jugador. calculatePoints() llama calculateCardPoints() para obtener los puntos de la carta en concreto y la suma al total. Si la carta es un as, incrementa un contador de ases, por lo que, si hay más de un as en la mano, resta 10 puntos a cada as para que el total sea menos de 21.

```

/**
 * Calculates the points for a single card.
 * @param card The card to calculate points for.
 * @return The points value of the card.
 */
private fun calculateCardPoints(card: Card): Int {
    return if (card.rank != Rank.ACE) {
        if (card.rank.ordinal >= Rank.JACK.ordinal) 10 else
        card.rank.ordinal + 1
    } else {
        card.maxPoints
    }
}

/**
 * Calculates the total points for a hand of cards.
 * @param hand The hand of cards to calculate points for.
 * @return The total points of the hand.
 */
fun calculatePoints(hand: List<Card>): Int {
    var total = 0
    var aces = 0

    // Calculate the total points, taking into account the value of
    aces
    for (card in hand) {
        total += calculateCardPoints(card)
        if (card.rank == Rank.ACE) {
            aces++
        }
    }

    // If the total is over 21 and there are aces, subtract 10 for
    each ace
    while (total > 21 && aces > 0) {
        total -= 10
        aces--
    }

    return total
}

```

El resto de funciones tienen un funcionamiento simple. La función `checkForBlackjack()`, comprueba si algún jugador tiene blackjack al inicio del juego. Si es así, declara ganador a ese jugador.

La función `restartGame()`, se encarga de reiniciar el juego, limpiando las manos de los jugadores y reseteando el estado del juego.

La función `toggleMusic()`, se encarga de activar y desactivar la música del juego. De nuevo añadí un tratamiento `try catch` en caso de que la música no pudiese ser reproducida.

```

private fun checkForBlackjack() {
    for (player in _players.value!!) {
        if (calculatePoints(player.hand) == 21) {
            _winner.value = player
        }
    }
}

```

```

        _showDialog.value = true
        return
    }
}

/**
 * Restarts the game by clearing the players' hands and resetting the
 * game state.
 */
private fun restartGame() {
    for (player in _players.value!!) {
        player.hand.clear()
    }
    _currentTurn.value = _players.value?.first()
    _gameInProgress.value = false
}

/**
 * Toggles the music on and off.
 * @param context The context to use to create the MediaPlayer.
 */
fun toggleMusic(context: Context) {
    try {
        if (isPlaying) {
            mediaPlayer?.stop()
            mediaPlayer?.release()
            mediaPlayer = null
            isPlaying = false
        } else {
            mediaPlayer = MediaPlayer.create(context, R.raw.blackjack)
            mediaPlayer?.start()
            isPlaying = true
        }
    } catch (e: Exception) {
        Log.e("BlackjackGameViewModel", "Error al reproducir la
música: ${e.message}")
    }
}

```

## BlackjackGame

En cuanto a la vista del modo 1vs1 no hay casi nada diferente. De nuevo llamo a las LiveData con observeAsState() y dejo toda la lógica de negocio en mi viewModel. Si bien es cierto que he intentado gestionar el diseño de la interfaz de distinta manera al intentar colocar las cartas en diagonal, al final el resultado es similar.

```

@Composable
fun BlackjackScreen(gameViewModel: BlackjackGameViewModel) {
    // Get the game state from the ViewModel
    val backgroundImage = painterResource(id = R.drawable.tapete)
    val players by gameViewModel.players.observeAsState(initial =
emptyList())
    val winner by gameViewModel.winner.observeAsState()
    val gameInProgress by
gameViewModel.gameInProgress.observeAsState(initial = false)
    val currentTurn by gameViewModel.currentTurn.observeAsState()

    // Display the game screen

```

```

Box(modifier = Modifier.fillMaxSize()) {
    Image(
        painter = backgroundImage,
        contentDescription = null,
        modifier = Modifier.fillMaxSize(),
        contentScale = ContentScale.FillBounds
    )

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.SpaceEvenly
    ) {
        Spacer(modifier = Modifier.height(16.dp))

        // Display the current turn
        currentTurn?.let {
            Text(text = "Turno: ${it.name}", color = Color.White)
        }
        Spacer(modifier = Modifier.height(16.dp))

        // Display the players' cards or the winner and start game
        button
        if (gameInProgress) {
            players.forEachIndexed { _, player ->
                currentTurn?.let { PlayerCard(player,
gameViewModel) }

```

La componible PlayerCard() es la que incluye la pantalla de juego del modo 1vs1. Aquí se incluye un Alert Dialog para determinar el ganador y otro para el empate.

```

@Composable
fun PlayerCard(player: Player, gameViewModel: BlackjackGameViewModel)
{
    // Get the winner from the ViewModel
    val winner by gameViewModel.winner.observeAsState()

    // Display a dialog if the game is over
    if (winner != null) {
        AlertDialog(
            onDismissRequest = { gameViewModel.closeDialog() },
            title = { Text(text = "Game Over") },
            text = { Text(text = "Winner is: ${winner!!.name}") },
            confirmButton = {
                Button(onClick = { gameViewModel.closeDialog() },
                    colors =
ButtonDefaults.buttonColors(containerColor = Color(0xFFD4AF37)),
                    border = BorderStroke(2.dp, Color.White)) {
                    Text("Aceptar",
                        color = Color.Black)
                }
            }
        )
    }

    } else if (gameViewModel.showDialog.value == true) {
        AlertDialog(
            onDismissRequest = { gameViewModel.closeDialog() },
            title = { Text(text = "Game Over") },

```

```

        text = { Text(text = "El resultado es de empate") },
        confirmButton = {
            Button(onClick = { gameViewModel.closeDialog() },
                colors =
                    ButtonDefaults.buttonColors(containerColor = Color(0xFFD4AF37)),
                border = BorderStroke(2.dp, Color.White)
            ) {
                Text("Aceptar")
            }
        }
    }
)
}

```

Como dije, este fue el primer modo sobre el que trabajé durante el proyecto. Por esto, quizás la forma en la que gestiono si una carta debe mostrarse bocabajo o no, quizás sea algo menos elegante, puesto que siempre que ha sido posible, he intentado evitar el uso de variables o lógica dentro de la vista.

Siendo el caso de que la Vista solo observa el estado del ViewModel y se actualiza con los cambios que allí se realizan, creo que sigue siendo correcto, puesto que al final la Vista no modifica el estado del juego, sino que toma decisiones sobre lo que va a mostrar en base a un estado u otro del viewModel.

```

player.hand.forEachIndexed { cardIndex, card ->
    val isGameOver = gameViewModel.winner.value != null ||
gameViewModel.showDialog.value == true
    val shouldHideCard = gameViewModel.currentTurn.value
!= player && cardIndex != 0 &&
        !isGameOver
    val cardResource = if (shouldHideCard) {
        R.drawable.bocabajo2
    } else {
        card.idDrawable
    }
    Image(
        painterResource(id = cardResource!!),
        contentDescription = null,
        modifier = Modifier
            .height(150.dp)
            .width(75.dp)
            .offset(x = (cardIndex * 15).dp, y =
(cardIndex * 20).dp)
    )
}

Spacer(modifier = Modifier.height(18.dp))

Row(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.Center
) {
    Button(
        onClick = { gameViewModel.hitMe(player) },
        colors = ButtonDefaults.buttonColors(containerColor =
Color(0xFFD4AF37)),
        border = BorderStroke(2.dp, Color.White)
    ) {
        Text(text = "Dame carta", color = Color.Black)
    }
}

```

```

    }

    Spacer(modifier = Modifier.width(8.dp))

    Button(
        onClick = { gameViewModel.pass(player) },
        colors = ButtonDefaults.buttonColors(containerColor =
Color(0xFFD4AF37)),
        border = BorderStroke(2.dp, Color.White)
    ) {
        Text(text = "Plantarse", color = Color.Black)
    }
}
}
}

```

## HighestCardViewModel

Este es el ViewModel dedicado a la lógica del juego de la carta más alta. Este juego de cartas es más sencillo que el Blackjack. Lo he podido desarrollar siguiendo la estructura MVVM con mayor agilidad que la primera vez que trabajé en él. El ViewModel se compone de la llamada al mapa de Imágenes como en los anteriores ViewModel:

```

**
 * ViewModel for the HighestCard game.
 *
* @property deck The deck of cards for the game.
* @property currentCard LiveData representing the current card in the
game.
* @property remainingCards LiveData representing the number of
remaining cards in the deck.
* @property isDeckReset LiveData representing whether the deck has
been reset.
*/
class HighestCardViewModel : ViewModel() {
    // The deck of cards
    private val deck = Deck(cardImageMap = mapOf("corazones" to
R.drawable.corazones,
        "corazones2" to R.drawable.corazones2,
        "corazones3" to R.drawable.corazones3,
        "corazones4" to R.drawable.corazones4,
        "corazones5" to R.drawable.corazones5,

```

A partir de ahí las LiveData que usaré para pasar la información a la Vista, más un bloque init que llama a la función resetGame(). resetGame baraja el mazo y cambia el estado de la variable booleana isDeckReset a true.

```

val currentCard = MutableLiveData<Card>()

// LiveData for the remaining cards in the deck
val remainingCards = MutableLiveData<Int>()

// LiveData for the deck state
val isDeckReset = MutableLiveData<Boolean>()

```

```

init {
    // Initialize the game
    resetGame()
}

/**
 * Resets the game.
 * Shuffles the deck, updates the remaining cards, sets the
 * current card to null, and updates the deck state.
 */
fun resetGame() {
    // Shuffle the deck
    deck.shuffle()

    // Update the remaining cards
    remainingCards.value = deck.cardsList.size

    // Set the current card to null
    currentCard.value = null

    // Update the deck state
    isDeckReset.value = true
}

/**
 * Handles the "Hit Me" action in the game.
 * Checks if the deck has cards, gets a card from the deck if
 * available, and updates the remaining cards.
 */
fun hitMe() {
    // Check if the deck has cards
    if (deck.hasCards()) {
        try {
            // Get a card from the deck
            currentCard.value = deck.getCard()

            // Update the remaining cards
            remainingCards.value = deck.cardsList.size
        } catch (e: Exception) {
            Log.e("BlackjackDealerViewModel", "Error obtaining a
card: ${e.message}")
        }
    }
}
}

```

## HighestCard

HighestCard es un archive kotlin que se compone de una única función componible, la cual, conforma mi pantalla para el juego de la Carta más alta. Aquí me he mantenido fiel al estilo original del resto del juego. Mismos botones y fondo de pantalla. Me he permitido hacer mayor el tamaño de las cartas por la sencillez del layout.

```

@Composable
fun HighestCardScreen(highestCardViewModel: HighestCardViewModel) {
    // Observe the current card
    val currentCard by
highestCardViewModel.currentCard.observeAsState()
}

```



```

        // Observe if game needs to be reset
        val isDeckReset by
highestCardViewModel.isDeckReset.observeAsState()

        // Load the background image
        val backgroundImage = painterResource(id = R.drawable.tapete)

        // Load the face down card image
        val faceDownImage = painterResource(id = R.drawable.bocabajo)

        // Create a Box to layer the background image and the game UI
        Box(modifier = Modifier.fillMaxSize()) {
            // Display the background image
            Image(
                painter = backgroundImage,
                contentDescription = null,
                modifier = Modifier.fillMaxSize(),
                contentScale = ContentScale.FillBounds
            )

            // Display the game UI
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .padding(16.dp),
                verticalArrangement = Arrangement.SpaceEvenly,
                horizontalAlignment = Alignment.CenterHorizontally
            ) {
                // Display the current card
                if (currentCard == null) {
                    // If there's no current card, show the face down card
                    Image(
                        painter = faceDownImage,
                        contentDescription = "Carta bocabajo",
                        modifier = Modifier.size(200.dp)
                    )
                } else {
                    // If there's a current card, show it
                    val cardDrawable = currentCard!!.idDrawable?.let {
painterResource(id = it) }
                    if (cardDrawable != null) {
                        Image(
                            painter = cardDrawable,
                            contentDescription = "Carta actual:
${currentCard!!.rank} de ${currentCard!!.suit}",
                            modifier = Modifier.size(200.dp)
                        )
                    }
                }

                // Display the remaining cards
                Text(text = "Cartas restantes:
${highestCardViewModel.remainingCards.value}",
                    color = Color.White)

                // Display the "Hit Me" button
                Button(
                    onClick = { highestCardViewModel.hitMe() },
                    colors = ButtonDefaults.buttonColors(containerColor =
Color(0xFFD4AF37)),
                    border = BorderStroke(2.dp, Color.White)
                )
            }
        }
    }
}

```

```

        ) {
            Text(text = "¡Dame carta!",
                color = Color.Black)
        }

        // Display the "Reset" button
        Button(
            onClick = { highestCardViewModel.resetGame() },
            colors = ButtonDefaults.buttonColors(containerColor =
Color(0xFFD4AF37)),
            border = BorderStroke(2.dp, Color.White)
        ) {
            Text(text = "    Reiniciar    ",
                color = Color.Black)
        }
    }
}
}

```

## MainMenu

Mi archivo MainMenu solo es un archivo kotlin con la inclusión de una sencilla componible que permite navegar a un lugar u a otro del juego a conveniencia.

La variable context que aquí uso, es para poder llevarme la música desde el viewModel, debido a que la música no está necesariamente vinculada al ciclo de vida del juego, no existen peligros de fugas de memoria por lo que he considerado correcto su inclusión.

```

fun MainMenu(navController: NavController, gameViewModel:
BlackjackGameViewModel) {
    // Remember a mutable state for the open dialog
    val openDialog = remember { mutableStateOf(false) }
    // Get the background image
    val background = painterResource(id = R.drawable.menu)
    // Get the current context
    val context = LocalContext.current
}

```

El botón de Salir y el botón de activar/desactivar la música son las únicas llamadas al viewModel en esta componible, quedando vacío de casi cualquier interés más allá del simple diseño de la misma, que he repetido en las distintas pantallas.

```

// Exit button
Button(
    onClick = { gameViewModel.closeApp() },
    colors = ButtonDefaults.buttonColors(containerColor =
Color(0xFFD4AF37)),
    border = BorderStroke(2.dp, Color.White),
    modifier = Modifier.sizeIn(minWidth = 200.dp,
minHeight = 50.dp)
) {
    Text("Salir", color = Color.Black)
}

// Music toggle button

```

```

        Box(modifier = Modifier.fillMaxSize(), contentAlignment =
Alignment.BottomEnd) {
            IconButton(onClick = { gameViewModel.toggleMusic(context) }) {
                Icon(painterResource(R.drawable.icmusicnote),
contentDescription = "Toggle Music")
            }
        }
    }
}

```

## Results

Results.kt es un archivo con la componible ResultsScreen(). ResultsScreen(), almacena la puntuación de los jugadores en el modo 1vs1.

```

/**
 * A composable function to display the results screen.
 * @param gameViewModel The ViewModel for the game.
 */
@Composable
fun ResultsScreen(gameViewModel: BlackjackGameViewModel) {
    val player1Wins by
gameViewModel.player1Wins.observeAsState(initial = 0)
    val player2Wins by
gameViewModel.player2Wins.observeAsState(initial = 0)

    Box(modifier = Modifier.fillMaxSize()) {
        Image(
            painter = painterResource(id = R.drawable.tapete),
            contentDescription = null,
            modifier = Modifier.fillMaxSize(),
            contentScale = ContentScale.FillBounds
        )
    }

    Column(
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.spacedBy(10.dp),
        modifier = Modifier.padding(16.dp)
    ) {
        Text(text = "Resultados 1vs1", color = Color.White, fontSize =
24.sp)
        Spacer(modifier = Modifier.height(16.dp))
        Text(text = "Victorias del Jugador 1: $player1Wins", color =
Color.White)
        Text(text = "Victorias del Jugador 2: $player2Wins", color =
Color.White)
    }
}

```

## MainActivity

El MainActivity se dedica a almacenar los objetos viewModels que he usado durante el proyecto. Dentro del onCreate() he añadido el NavHost con las navegaciones correspondientes a los objects BlackjackRoutes que aparecen en el principio de esta guía, con esto garantizo una navegación satisfactoria entre las pantallas del programa

```

/**
 * The main activity of the Blackjack game.
 */
class MainActivity : ComponentActivity() {
    // ViewModel for the vs game
    private val vsGameViewModel: BlackjackGameViewModel by
viewModels()

    // ViewModel for the dealer game
    private val dealerGameViewModel: BlackjackDealerViewModel by
viewModels()

    // ViewModel for the Highest Card game
    private val highestCardViewModel : HighestCardViewModel by
viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            BlackjackSpecialTheme {
                // A surface container using the 'background' color
from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    val navController = rememberNavController()

                    NavHost(
                        navController = navController,
                        startDestination =
BlackjackRoutes.MainMenuScreen.route
                    ) {

                        composable(BlackjackRoutes.MainMenuScreen.route) {
                            MainMenu(navController = navController,
gameViewModel = vsGameViewModel)
                        }

                        composable(BlackjackRoutes.BlackjackScreen.route) {
                            BlackjackScreen(gameViewModel =
vsGameViewModel)
                        }

                        composable(BlackjackRoutes.BlackjackDealerScreen.route) {
                            BlackjackDealerScreen(blackjackDealerViewModel = dealerGameViewModel)
                        }

                        composable(BlackjackRoutes.ResultsScreen.route) {
                            ResultsScreen(gameViewModel =
vsGameViewModel)
                        }

                        composable(BlackjackRoutes.HighestCardScreen.route) {
                            HighestCardScreen(highestCardViewModel =
highestCardViewModel)
                        }
                    }
                }
            }
        }
    }
}

```

```

    }
}
}

```

Por último, simplemente añado una mención al cierre de la app.

```

// Observe the event to close the app
vsGameViewModel.eventCloseApp.observe(this) { event ->
    if (event) {
        finish()
        vsGameViewModel.onAppClosed()
    }
}

```

## Conclusiones

Este trabajo ha sido muy satisfactorio ya que ha puesto a prueba mis capacidades de programación y de resolución de problemas. Me siento satisfecho con el trabajo realizado, además, elaborando la guía he podido ver las diferencias entre el código con el que empecé a trabajar hace varias semanas y el código más reciente, viendo una evolución en mi forma de razonar y entender el código. Creo que el ejemplo perfecto es mi lógica para tapar/destapar las cartas de la mesa en contraste con aquella para tapar las cartas del jugador que no ostenta el turno en el 1vs1.

Durante el proyecto también he cometido fallos al obnubilarme intentando ir más allá de lo que era razonable en una fase primaria o primera etapa, no obstante, a veces, reiniciar y cambiar el enfoque, garantiza una mejor construcción y una mejor visión del contexto. Me quedo con las ganas de añadir una sección de apuestas, y una serie de jugadores controlados por la IA que pueden dejarte sin dinero, estos tendrían su propio dinero para apostar. Creo que hubiese sido interesante añadir 5 rivales distintos, y que la misión del jugador hubiese sido arruinar a los 5, lo que lo establece como ganador del juego.