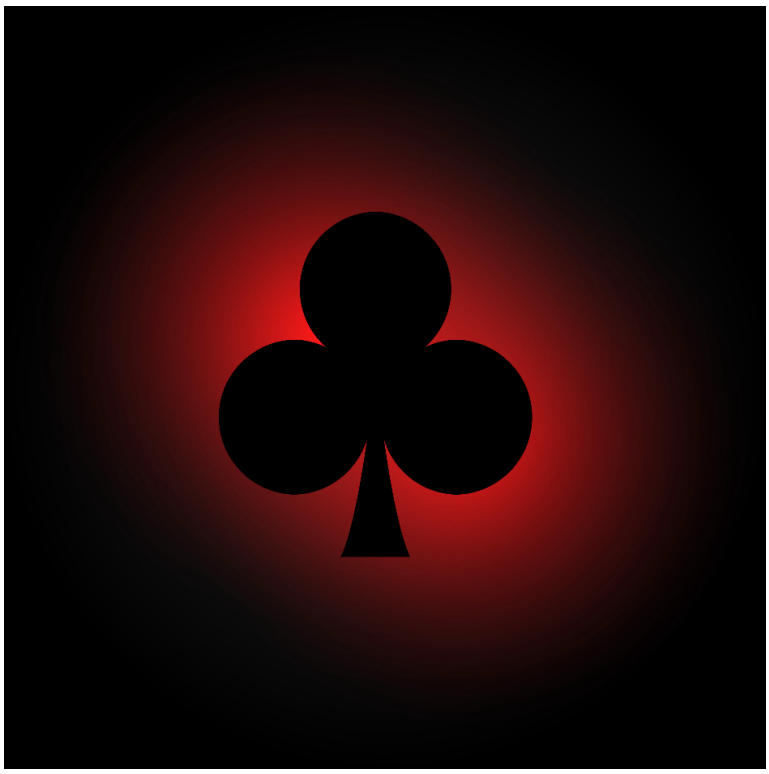
Práctica Final: Blackjack



Desarrollo de Aplicaciones Avanzadas (DAA)

VNiVERSiDAD D SALAMANCA

Shaiel Villalón Antolín 71042201P

ÍNDICE:

1. INTRODUCCIÓN

Para esta práctica final y, para ser más explícito, como se ha pedido en clase, sin hacer un informe muy largo, he explicado lo primero las vistas de la aplicación, dando por hecho las conexiones entre los botones y las pantallas ayudándome del 'NavigationController' para hacer que la navegación sea cotidiana y, a partir de ello, he explicado solamente los elementos más importantes de la práctica, divididos en carpetas, tal y cómo se pide, ya que se debe usar el MVVM.

Dentro de las carpetas he ido archivo por archivo explicando lo más relevante de cada uno de ellos, para entender la lógica y el funcionamiento final de la práctica

2. EXPLICACIÓN DE LAS VISTAS DE UN PROYECTO

Para empezar, se llevará a cabo la explicación de las diferentes vistas del proyecto, las funciones que llevan a cabo y los requisitos que cumplen.

Vistas del proyecto y explicación de ellas:

1- "StartViewController" o pantalla de inicio:

Es la pantalla principal de la app; la función específica que cumple es actuar como menú, a partir del cual el usuario puede acceder a todo el contenido de la aplicación. Es la base del flujo de navegación, al pulsar uno u otro botón va a la pantalla con la funcionalidad requerida.

En esta pantalla se muestra el título del juego junto con una imagen/logo (ambos centros con 'constraints') y tres botones centrados (también con 'constraints'). Se usa 'NavigationController' para que la navegación entre vistas sea natural, tal y como pide el enunciado de la práctica.

2- "GameViewController" o pantalla de juego

Es la pantalla donde se desarrolla la parte más importante de la aplicación, es decir, toda la partida de 'Blackjack'.

En ella, se muestran las cartas del jugador y del 'bot' en dos 'UIStackView' diferentes, también, encima del 'StackView' del 'bot' se muestra su puntuación en cada momento y, debajo del del jugador, la puntuación de este.

Además, en la parte inferior de esta pantalla se colocan tres botones, uno para empezar una nueva partida y otros dos para pedir una nueva carta o plantarse.

En esta pantalla se muestran las cartas del juego (imágenes de la API), las puntuaciones y, al finalizar la partida, se ejecuta la lógica del juego y se muestra una pantalla modal para introducir el nombre del jugador, que muestra quién ha ganado la partida. Por último, al darle al botón de "Guardar" de la pantalla modal, se guarda el resultado en la pantalla del historial de la aplicación.

3- “HistoryViewController” o pantalla del historial

Es la pantalla que muestra todas las partidas jugadas y guardadas, ordenadas cronológicamente, la de más arriba es la más actual y la de más debajo de todas es la más antigua.

Se utiliza un ‘UITableView’ para listar las partidas, mostrando los datos necesarios y, además, al seleccionar una celda (una partida guardada en el historial) se abre en detalle dicha partida.

4- “HistoryDetailViewController” o pantalla detalle del historial

Es la pantalla que muestra el detalle de una de las partidas seleccionadas en la pantalla anterior (‘HistoryViewController’).

En ella, se muestra el nombre del jugador, la fecha en que se jugó la partida, un emoji que simboliza el resultado de la partida y la puntuación final tanto del jugador como del ‘bot’.

5- “OptionsViewController” o pantalla de opciones

Esta pantalla permite activar o desactivar la música del juego con la utilización de un ‘switch’.

Dicho interruptor, controla la música de la aplicación, que se reproduce usando ‘AVAudioPlayer’, tal y como se pide en el enunciado de la práctica. Además, se hace uso de las notificaciones cuando se activa o desactiva la música.

También se usa una ‘UIView’ para simular los colores del tablero (usados en todas las pantallas).

3. CARPETA ‘MODEL’

Dentro de la carpeta ‘Model’ y siguiendo el patrón MVVM, se encuentran los archivos ‘Card.swift’, ‘GameResult.swift’ y ‘GameSession.swift’.

En el primero de ellos, se incluyen dos estructuras; la primera de ellas representa una carta real, obtenida de la API y la segunda modela las respuestas completas que devuelve la API (‘success’ indica si la petición fue correcta y, las siguientes variables, guardan el identificador del mazo, el número de cartas restantes del mazo, si el mazo está o no barajado y el array de cartas recibidas)

```swift
import Foundation

struct Card: Codable {
    let code: String
    let image: String
    let value: String
    let suit: String
}


struct DeckResponse: Codable {
    let success: Bool
    let deck_id: String
    let remaining: Int
    let shuffled: Bool?
    let cards: [Card]?
}
```

El segundo archivo tiene como objetivo guardar los datos de una partida, es decir, cuando una partida se termina, se crea un objeto ‘GameResult’ que guarda datos de la partida (nombre, puntuaciones, resultado y fecha).

```swift
import Foundation

struct GameResult: Codable {
    let playerName: String
    let playerScore: Int
    let botScore: Int
    let result: String
    let date: Date
}
```

El tercer y último archivo de la carpeta ‘Model’ (‘GameSession.swift’), es una clase ‘Singleton’, que almacena en memoria los datos de una partida. ‘GameResult’ y ‘GameSession’ no son lo mismo, aunque parezca que ambos guardan la información, el primero de ellos guarda los datos de la partida y, el segundo (el de la explicación actual), guarda dichos datos en la memoria.

El ‘Singleton’ es tal cual se ha proporcionado en el enunciado de la práctica, lo único que se ha añadido es el campo ‘playerName’, que guarda el último nombre que el usuario introdujo como nombre del jugador.

```swift
import Foundation

final class GameSession {
    static let shared = GameSession()
    private init () {}

    //Historial
    var history: [GameResult] = []

    var playerName: String = "Jugador"
}
```

4. CARPETA ‘VIEWMODEL’

Dentro de la carpeta ‘ViewModel, se encuentran los archivos ‘GameViewModel.swift’, ‘OptionsViewModel.swift’ y ‘HistoryViewModel.swift’.

Contiene la lógica de la aplicación, procesando datos, llamando a la API o decidiendo qué mostrar en la interfaz, sin dibujar nada.

‘GameViewModelswift’ se encarga de gestionar las acciones de la aplicación, siendo el “cerebro” del juego; esto quiere decir que la vista (‘GameViewController.swift’) no tiene lógica del juego, solo usa métodos de este archivo (se explicará en detalle lo más importante de este archivo).

Dentro del archivo, se crean variables privadas internas no accesibles desde la vista y variables públicas para que la vista pueda leer las cartas.

Se crean las siguientes variables que permiten que el ‘ViewModel’ comunique cambios sin conocer la vista, para que esta se actualice:

```
var updateCard: (() -> Void)?
var updateStatus: ((String) -> Void)?
var endedGame: ((GameResult) -> Void)?
```

Para iniciar la partida, se limpian las manos y el estado y se crea un nuevo mazo (siguiente imagen). Primero se construye la URL y se guarda en una variable, se llama a la API y, al recibir el resultado, se guarda el id del mazo creado y se inicia el reparto inicial.

```
private func createNewDeck () {
    guard let url = URL(string:
        "https://deckofcardsapi.com/api/deck/new/shuffle/?deck_count=1") else {
        updateStatus?("Url invalida")
        return
    }

    let task = URLSession.shared.dataTask(with: url) { data, _, error in
        guard let data = data, error == nil else { return }
        do {
            let deckResponse = try JSONDecoder().decode(DeckResponse.self, from: data)
            self.deckID = deckResponse.deck_id
            self.initialDeal ()
        } catch {
            print ("Error al crear el mazo")
        }

    }
    task.resume()
}
```

En la función del reparto, se piden dos cartas del mazo y se reparten una al jugador y otra al 'bot', de esta manera se evita que la carta sea la misma (si se hacen dos llamadas, puede ocurrir que se robe la misma carta), después, avisa a la vista para actualizar cartas.

La pedir las cartas como se dice anteriormente, se usar la función mostrada en la siguiente imagen, se hace la llamada a la URL, se decodifican las cartas y se llama al 'completion' con las cartas.

```
private func drawCards (count: Int, completion: @escaping ([Card]?) -> Void) {

    guard !deckID.isEmpty else { completion(nil);return }
    let url = URL(string:
        "https://deckofcardsapi.com/api/deck/\(deckID)/draw/?count=\(count)")!
    URLSession.shared.dataTask(with: url) { data, _, error in
        guard let data = data, error == nil else {completion(nil); return }
        do {
            let respuesta = try JSONDecoder().decode(DeckResponse.self, from: data)
            completion(respuesta.cards)
        } catch {
            print ("Error al mostrar cartas")
            completion(nil)
        }

    }.resume()

}
```

También, en este archivo, se añade la lógica del 'bot' para plantarse (si tiene puntuación igual o mayor a 17) o pedir otra carta calculando dicha puntuación con otras funciones complementarias, que ponen la 'J', dama y rey con valor 10, el 'as' con valor 11 (pasará a ser valor 1 si se ha superado la puntuación igual a 21) y que calculan la puntuación final hasta el momento.

Por último, se determina el resultado con otra función, comprobando si ya terminó la partida, comparando resultado, creando un 'GameResult' y notificando a la vista para que se actualice.

El archivo 'OptionsViewModel.swift' se encarga de gestionar toda la lógica de la música, su reproducción, su pausa, las notificaciones y el interruptor. Primero, se crean propiedades y se crea el método 'toggle' (siguiente imagen), que cambia el estado y, si se pasa al estado 'on', se llama a la función de reproducir música y envía una notificación local y, si se pasa de 'on' a 'off', se llama a la función para parar la música y se envía también una notificación local.

```
func toggle() {
    onMusic.toggle()
    if onMusic {
        startMusic()
        sendMusicNotification(activated: true)
    } else {
        stopMusic()
        sendMusicNotification(activated: false)

    }
}
```

La función de iniciar la música reproduce “background.mp3” (cargada anteriormente) en bucle y la función ‘stopMusic()’ para la música.

Por último, la función ‘sendMusicNotification()’, notifica al usuario de la situación de la música, si se activa o desactiva.

Para terminar con la lógica, el último archivo, ‘HistoryViewModel.swift’, controla lo relacionado con el historial de las partidas jugadas y guardadas. En este caso, devuelve el historial ordenado por fecha descendente usando:

```
var history: [GameResult] {
    return GameSession.shared.history.sorted { $0.date > $1.date }
}
```

También, la otra función de este archivo es la de la siguiente imagen, esta función elimina todo el historial de partidas guardadas en la aplicación; se llama a esta función desde ‘HistoryViewController.swift’, se explicará más adelante.

```
func clearHistory () {
    GameSession.shared.history.removeAll()
}
```

5. CARPETA ‘VIEW’

Dentro de la carpeta ‘View, se encuentran los archivos ‘StartViewController.swift’, ‘GameViewController.swift’, ‘OptionsViewController.swift’, ‘HistoryViewController’ y ‘HistoryDetailViewController’.

Las vistas muestran los datos provenientes del ‘ViewModel’, reaccionan a los cambios mediante los ‘callbacks’ creados y explicados anteriormente:

```
var updateCard: (() -> Void)?
var updateStatus: ((String) -> Void)?
var endedGame: ((GameResult) -> Void)?
```

Solo gestionan interfaz y navegación, ni realizan cálculos ni contienen lógica de negocio.

La vista de inicio, ‘StartViewController.swift’, muestra lo explicado en el punto 2 y, además de eso, envía datos a ‘GameViewController’ mediante la llamada que se encuentra dentro de la función que se muestra en la siguiente imagen:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    if segue.identifier == "showGame",
       let destination = segue.destination as? GameViewController {


        destination.playerNameValue = GameSession.shared.playerName
    }
}
```

La vista del juego, ‘GameViewController.swift’, como he explicado en el punto 2, es la pantalla donde se juega la partida, mostrando las cartas de ambos jugadores, se muestran los botones para empezar partida, plantarse y pedir una nueva carta y, también, se muestra el resultado.

Una de las funciones más importantes de este archivo es la siguiente, que dice al ‘ViewModel’ cómo avisar a la vista cuando cambian cartas, estado y cuando el juego termina.

Con esto, se actualizan cartas, se muestran mensajes del ganador de la partida y se presenta el cuadro modal de que se acabó la partida.

```
private func setupBindings() {
    viewModel.updateCard = { [weak self] in
        guard let self = self else { return }
        DispatchQueue.main.async {
            self.updateCards(for: self.playerStack, with: self.viewModel.playerCardsPublic)
            self.updateCards(for: self.botStack, with: self.viewModel.botCardsPublic)
            self.updateScore()
        }
    }

    viewModel.updateStatus = { [weak self] message in
        DispatchQueue.main.async {
            self?.status.text = message
        }
    }

    viewModel.endedGame = { [weak self] res in
        guard let self = self else { return }
        DispatchQueue.main.async {
            self.requestCardButton.isEnabled = false
            self.standButton.isEnabled = false
            self.newgameButton.isEnabled = true

            self.showResModal(for: res)
        }
    }
}
```

Desde este archivo, también, se cargan y se muestran las imágenes de las cartas en los 'StackView' y, pulsando los botones de la pantalla, se llama a 'ViewModel' para llevar a cabo las acciones que muestra el botón.

La vista de opciones, 'OptionsViewController.swift', además de lo explicado en el punto 2, lo importante de este archivo es la sincronización del switch con el 'ViewModel', la llamada al 'toggle()' y el requisito de notificaciones, implementado de la siguiente manera:

```
private func requestNotification() {
    let center = UNUserNotificationCenter.current()
    center.requestAuthorization(options: [.alert, .sound, .badge]) { granted, error in

        if granted {
            print ("Permiso concedido")
        } else {
            print("Permiso denegado")
        }

    }
}
```

Por último, las vistas 'HistoryViewController.swift' y 'HistoryDetailViewController.swift', a parte de lo explicado en el punto 2, como en los anteriores archivos, tienen parte de código importante para la completa implementación de la práctica.

En el primer archivo de los dos nombrados anteriormente, utilizamos la función siguiente porque, cada vez que la vista del historial aparece, necesitamos recargar la tabla.

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(animated)
    tableView.reloadData()
}
```

Una de las acciones opcionales que se han añadido es la de borrar el historial de las partidas, para ello, se ha implementado la función de la siguiente imagen, que crea una ventana emergente para evitar errores de darle sin querer a la papelera y, después, si el botón pulsa la opción “Borrar”, llama a ‘self.ViewModel.clearHistory()’ de ‘HistoryViewModel’ y, ahí, las borra; después de ello, vuelve a carga la tabla. El botón “Cancelar”, si se pulsa, no se borra nada y se cierra la alerta.

```
@IBAction func clearHistoryTapped(_ sender: UIBarButtonItem) {
    let alert = UIAlertController(
        title: "Borrar historial",
        message: "¿Seguro que quieres borrar todo el historial?",
        preferredStyle: .alert
    )
    alert.addAction(UIAlertAction(title: "Borrar", style: .destructive) { _ in
        self.viewModel.clearHistory()
        self.tableView.reloadData()
    })
    alert.addAction(UIAlertAction(title: "Cancelar", style: .cancel))
    present(alert, animated: true)
}
```

También usamos las siguientes dos funciones para (la primera de ellas) devolver el número de celdas y, la segunda de la siguiente imagen, obtiene ‘GameResult’ desde ‘ViewModel’, configura las etiquetas y ajusta los márgenes.

```swift
func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return viewModel.history.count
}

func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

    let r = viewModel.history[indexPath.row]
    let cell = tableView.dequeueReusableCell(withIdentifier: "HistoryCell", for: indexPath)


    cell.textLabel?.text = "\(emoji(for: r.result))  \(r.playerName)"
    cell.textLabel?.font = UIFont.boldSystemFont(ofSize: 20)
    cell.textLabel?.textColor = .white

    let formatter = DateFormatter()
    formatter.dateStyle = .medium
    formatter.timeStyle = .short

    let formattedDate = formatter.string(from: r.date)

    cell.detailTextLabel?.text = "\(formattedDate) — Jugador: \(r.playerScore) | Bot: \(r.botScore)"
    cell.detailTextLabel?.textColor = .white

    cell.detailTextLabel?.font = UIFont.systemFont(ofSize: 16)
    cell.contentView.layoutMargins = UIEdgeInsets(top:10, left: 15, bottom: 10, right: 15)

    return cell
}
```

También, usamos la función de la siguiente imagen para pasar el ‘GameResult’ al destino.

```swift
func tableView(_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {
    performSegue(withIdentifier: "showDetail", sender: indexPath)
    tableView.deselectRow(at: indexPath, animated: true)
}
```

En el archivo detalle del historial, se recibe un ‘GameResult’ desde el segue y se muestra la información requerida, además de decidir qué texto mostrar dependiendo del resultado.

6. VISTA FINAL

Para la vista final, en vez de enseñar y explicar cada vista una a una, se adjunta un link para poder ver el vídeo en caso de que el adjunto, junto con el proyecto de 'XCode', no se cargue correctamente.

https://drive.google.com/file/d/1DrhtjJse8FZVuYXdHrFnuclUTpeu2nf8/view?usp=drive_link