Unidad 4

Rodriguez Segura Fernando

Actividad 1

1 Resumen

Unscramble es una app de Android desarrollada con **Jetpack Compose** y escrita en **Kotlin**. El objetivo del jugador es ordenar las letras de una palabra desordenada correctamente.

2 Componentes Clave

- **ViewModel:** Maneja el estado del juego.
- **StateFlow:** Expone datos reactivos al UI.
- **UI (Jetpack Compose):** Interfaz declarativa y reactiva.

3 GameViewModel.kt

Listing 1: Clase GameViewModel.kt

```
class GameViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(GameUiState())
    val uiState: StateFlow<GameUiState> = _uiState

    private var usedWords = mutableSetOf<String>()
    private lateinit var currentWord: String

    init {
        resetGame()
    }

    fun resetGame() {
        usedWords.clear()
        pickNewWord()
        _uiState.value = GameUiState(
            score = 0,
            currentWordCount = 1,
            isGuessedWordWrong = false,
```

```
            isGameOver = false,
            scrambledWord = shuffleWord(currentWord)
        )
    }

    fun updateUserGuess(guess: String) {
        _uiState.value = _uiState.value.copy(userGuess = guess)
    }

    fun checkUserGuess() {
        val userGuess = _uiState.value.userGuess
        if (userGuess.equals(currentWord, ignoreCase = true)) {
            val updatedScore = _uiState.value.score +
               SCORE_INCREASE
            proceedToNextWord(updatedScore)
        } else {
            _uiState.value = _uiState.value.copy(isGuessedWordWrong
                = true)
        }
    }
}
```

Explicación:

- $_u iStateguardaelestadoactual$.`resetGame()`$reiniciaeljuego$.
- `checkUserGuess()` verifica si la palabra ingresada es correcta.

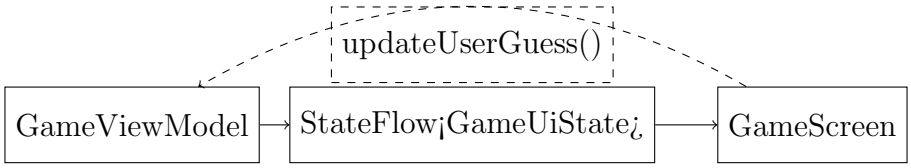
4 UI Principal (GameScreen.kt)

Listing 2: Composable GameScreen

```
@Composable
fun GameScreen(
    gameUiState: GameUiState,
    onUserGuessChanged: (String) -> Unit,
    onSubmitWord: () -> Unit
) {
    Column {
        Text(text = "Scrambled Word: ${gameUiState.scrambledWord}")
        OutlinedTextField(
            value = gameUiState.userGuess,
            onValueChange = onUserGuessChanged
        )
        Button(onClick = onSubmitWord) {
            Text("Submit")
        }
        if (gameUiState.isGuessedWordWrong) {
```

```
            Text("Incorrect guess!", color = Color.Red)
        }
    }
}
```

5 Flujo de Datos



6 Resumen General

Unscramble es una aplicación didáctica en Android donde el jugador debe adivinar palabras a partir de letras mezcladas. Se utiliza **Jetpack Compose** para la UI y **StateFlow + ViewModel** para manejar estado de forma reactiva.

7 Estructura del Proyecto

- **GameViewModel.kt**: Lógica del juego (palabras, puntaje, control de flujo).
- **GameUiState.kt**: Modelo que representa el estado actual de la UI.
- **GameScreen.kt**: UI declarativa en Jetpack Compose.
- **MainActivity.kt**: Entry point de la app, conecta el ViewModel con la UI.

8 Modelo de Estado: GameUiState.kt

Listing 3: Modelo de estado

```
data class GameUiState(
    val currentScrambledWord: String = "",
    val isGuessedWordWrong: Boolean = false,
    val score: Int = 0,
    val currentWordCount: Int = 0,
    val isGameOver: Boolean = false
)
```

Explicación: Este modelo es utilizado por el ViewModel para exponer los datos actuales del juego hacia la UI mediante un `StateFlow`.

9 Lógica del Juego: GameViewModel.kt

Listing 4: Inicialización del estado

```
class GameViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(GameUiState())
    val uiState: StateFlow<GameUiState> = _uiState

    init {
        resetGame()
    }

    fun resetGame() {
        ...
        pickNewWordAndShuffle()
    }
}
```

- Usa `MutableStateFlow` para exponer cambios reactivos.
- La función `resetGame()` reinicia el juego.
- Las palabras se almacenan en una lista privada y se seleccionan aleatoriamente.

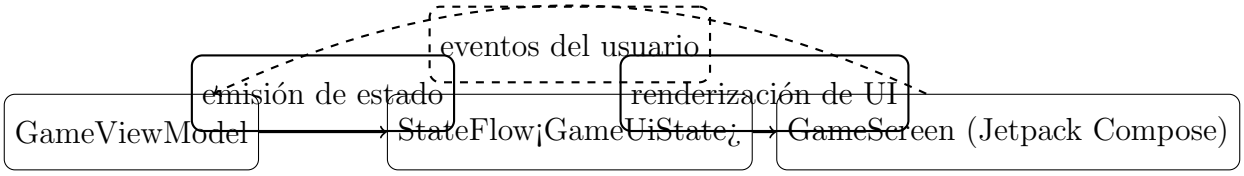
10 Interfaz: GameScreen.kt

Listing 5: UI con Jetpack Compose

```
@Composable
fun GameScreen(...) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Scrambled: $scrambledWord")
        OutlinedTextField(value = userGuess, onValueChange = { ...
           })
        Button(onClick = onSubmit) {
            Text("Submit")
        }
    }
}
```

Explicación: La interfaz muestra la palabra mezclada, un campo para adivinar y botones para enviar o pasar palabra. Se alimenta directamente del estado expuesto por el ViewModel.

11 Diagrama de Flujo de Datos



12 Introducción

El proyecto **Dessert Clicker** es una aplicación Android desarrollada en Kotlin usando Jetpack Compose. El objetivo del juego es "vender postres" haciendo clic en una imagen, acumulando ganancias ficticias.

13 Estructura del Proyecto

El proyecto está organizado de la siguiente forma:

- `MainActivity.kt`: Entrada principal de la app.
- `DessertClickerApp.kt`: Composable principal con navegación.
- `DessertClickerViewModel.kt`: Lógica de negocio (ViewModel).
- `ui/`: Componentes visuales y temas.

14 MainActivity

Este archivo inicia la app y registra eventos del ciclo de vida.

Listing 6: MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            DessertClickerApp()
        }
        Log.d("DessertClicker", "onCreate llamado")
    }
}
```

15 Vista principal

La vista muestra la imagen del postre actual y el contador de postres vendidos y ganancias.

Listing 7: DessertClickerApp.kt

```
@Composable
fun DessertClickerApp(viewModel: DessertClickerViewModel =
   viewModel()) {
    val uiState = viewModel.uiState.collectAsState().value
    DessertClickerScreen(
        revenue = uiState.revenue,
        dessertsSold = uiState.dessertsSold,
        dessertImageId = uiState.currentDessert.imageId,
        onDessertClicked = { viewModel.onDessertClicked() }
    )
}
```

16 ViewModel

Controla el estado del juego: número de postres vendidos, ganancias y lógica para cambiar de imagen.

Listing 8: DessertClickerViewModel.kt

```
class DessertClickerViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(DessertUiState())
    val uiState: StateFlow<DessertUiState> = _uiState

    fun onDessertClicked() {
        val currentState = _uiState.value
        val newRevenue = currentState.revenue + currentState.
           currentDessert.price
        val dessertsSold = currentState.dessertsSold + 1
        _uiState.value = currentState.copy(
            revenue = newRevenue,
            dessertsSold = dessertsSold,
            currentDessert = determineDessert(dessertsSold)
        )
    }
}
```

Actividad 2

17 Introducción

Cupcake App es una aplicación Android creada con Jetpack Compose y escrita en Kotlin. Guía al usuario a través del proceso de realizar un pedido de cupcakes, aplicando navegación entre pantallas y manejo de estado con ViewModel y StateFlow.

18 Objetivos del Proyecto

Este proyecto tiene como propósito:

- Introducir la navegación con Jetpack Compose.
- Manejar el estado con `ViewModel` y `StateFlow`.
- Implementar múltiples pantallas con datos compartidos.
- Compartir contenido con otras aplicaciones (share intent).

19 Estructura del Proyecto

- **MainActivity.kt** – Punto de entrada de la aplicación.
- **CupcakeApp.kt** – Define la navegación.
- **OrderViewModel.kt** – Contiene la lógica de estado del pedido.
- **Pantallas composables**:
  - StartOrderScreen
  - FlavorScreen
  - PickupScreen
  - SummaryScreen

20 ViewModel: OrderViewModel.kt

Listing 9: Ejemplo de ViewModel

```
class OrderViewModel : ViewModel() {

    private val _uiState = MutableStateFlow(OrderUiState())
    val uiState: StateFlow<OrderUiState> = _uiState

    fun setQuantity(quantity: Int) {
        _uiState.value = _uiState.value.copy(quantity = quantity)
        updatePrice()
    }

    fun setFlavor(flavor: String) {
        _uiState.value = _uiState.value.copy(flavor = flavor)
    }

    fun resetOrder() {
        _uiState.value = OrderUiState()
```

```
    }

    private fun updatePrice() {
        val price = _uiState.value.quantity * PRICE_PER_CUPCAKE
        _uiState.value = _uiState.value.copy(price = price)
    }
}
```

Descripción: Esta clase gestiona el estado del pedido: cantidad, sabor, fecha y precio total. Expone los datos como `StateFlow` para que la UI escuche los cambios.

21 Navegación: CupcakeApp.kt

Listing 10: Estructura de navegación

```
@Composable
fun CupcakeApp(viewModel: OrderViewModel = viewModel()) {
    val navController = rememberNavController()

    NavHost(navController = navController, startDestination = "
       start") {
        composable("start") {
            StartOrderScreen(onNext = { navController.navigate("
               flavor") })
        }
        composable("flavor") {
            FlavorScreen(onNext = { navController.navigate("pickup
               ") })
        }
        composable("pickup") {
            PickupScreen(onNext = { navController.navigate("summary
               ") })
        }
        composable("summary") {
            SummaryScreen(onCancel = {
                viewModel.resetOrder()
                navController.popBackStack("start", inclusive =
                   false)
            })
        }
    }
}
```

Descripción: Define las rutas de navegación entre pantallas usando `NavHost` de Jetpack Compose Navigation.

22 Pantallas: Ejemplo de FlavorScreen.kt

Listing 11: FlavorScreen.kt

```
@Composable
fun FlavorScreen(
    options: List<String>,
    onNext: () -> Unit,
    onSelect: (String) -> Unit
) {
    Column {
        Text("Selecciona un sabor")
        options.forEach { flavor ->
            Button(onClick = { onSelect(flavor) }) {
                Text(flavor)
            }
        }
        Button(onClick = onNext) {
            Text("Siguiente")
        }
    }
}
```

Descripción: Muestra una lista de sabores disponibles. Al seleccionar uno, se actualiza el estado y se permite avanzar.

23 Estado de UI: OrderUiState.kt

Listing 12: Modelo de UI

```
data class OrderUiState(
    val quantity: Int = 0,
    val flavor: String = "",
    val pickupDate: String = "",
    val price: String = ""
)
```

Este modelo representa el estado visual de la orden y es utilizado por el ViewModel y observado por la UI.

24 Interacción con otras Apps

La pantalla de resumen permite compartir el pedido por medio de un `Intent` de tipo "share":

Listing 13: Share Intent en SummaryScreen.kt

```
val context = LocalContext.current
val intent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
```

```
    putExtra(Intent.EXTRA_TEXT, orderSummary)
}
context.startActivity(Intent.createChooser(intent, "Compartir
   pedido"))
```

25 Introducción

La aplicación **Lunch Tray** es un proyecto de ejemplo desarrollado con **Jetpack Compose** y escrito en **Kotlin**. Su propósito es enseñar a los desarrolladores cómo implementar la navegación entre pantallas y gestionar el estado de la aplicación utilizando `ViewModel` y `StateFlow`.

26 Características Principales

- Navegación entre múltiples pantallas usando Jetpack Navigation Compose.
- Gestión del estado del pedido mediante ViewModel.
- Interfaz de usuario reactiva y declarativa con Compose.
- Cálculo dinámico del precio total del pedido.

27 Estructura del Proyecto

- `MainActivity.kt`: Entrada principal de la aplicación.
- `LunchTrayApp.kt`: Composable principal con lógica de navegación.
- `OrderViewModel.kt`: Lógica del pedido y gestión del estado.
- `ui/screens/`: Contiene las pantallas individuales (start, entree, side dish, accompaniment, checkout).
- `data/`: Contiene datos como precios y descripciones de los alimentos.

28 MainActivity

Listing 14: MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            LunchTrayApp()
        }
    }
}
```

29 Navegación: LunchTrayApp.kt

Listing 15: LunchTrayApp.kt

```
@Composable
fun LunchTrayApp() {
    val navController = rememberNavController()
    NavHost(navController = navController, startDestination = "
       start") {
        composable("start") { StartOrderScreen(...) }
        composable("entree") { EntreeMenuScreen(...) }
        composable("side") { SideDishMenuScreen(...) }
        composable("accompaniment") { AccompanimentMenuScreen(...)
           }
        composable("checkout") { CheckoutScreen(...) }
    }
}
```

Explicación: Se define un flujo de navegación entre pantallas usando rutas.

30 Manejo de Estado: OrderViewModel.kt

Listing 16: OrderViewModel.kt

```
class OrderViewModel : ViewModel() {
    private val _uiState = MutableStateFlow(OrderUiState())
    val uiState: StateFlow<OrderUiState> = _uiState

    fun setEntree(entree: String) {
        _uiState.update { it.copy(entree = entree) }
    }

    fun setSideDish(sideDish: String) {
        _uiState.update { it.copy(sideDish = sideDish) }
    }

    fun setAccompaniment(accompaniment: String) {
        _uiState.update { it.copy(accompaniment = accompaniment) }
    }

    fun resetOrder() {
        _uiState.value = OrderUiState()
    }
}
```

Explicación: Se usa StateFlow para exponer el estado del pedido a la interfaz de usuario. Cada pantalla modifica este estado.

31 Ejemplo de Pantalla: EntreeMenuScreen.kt

Listing 17: EntreeMenuScreen.kt

```
@Composable
fun EntreeMenuScreen(
    onNextClicked: () -> Unit,
    onCancelClicked: () -> Unit,
    onEntreeSelected: (String) -> Unit,
    options: List<String>
) {
    Column {
        Text("Choose an entree")
        options.forEach { entree ->
            Button(onClick = { onEntreeSelected(entree) }) {
                Text(entree)
            }
        }
        Button(onClick = onNextClicked) { Text("Next") }
        Button(onClick = onCancelClicked) { Text("Cancel") }
    }
}
```

32 Resumen de Funcionalidad

- El flujo va de `start → entree → side dish → accompaniment → checkout`.
- Cada pantalla actualiza el ViewModel.
- En la pantalla final, se muestra un resumen del pedido con el precio total.

Actividad 3

33 Introducción

La aplicación **Reply** es un cliente de correo electrónico básico desarrollado en **Kotlin** utilizando **Jetpack Compose**. Este proyecto forma parte del curso *Android Basics with Compose* ofrecido por Google Developer Training.

El propósito principal de **Reply** es demostrar el uso de diseños adaptativos en Jetpack Compose, permitiendo que la interfaz de usuario se ajuste dinámicamente a diferentes tamaños y tipos de pantalla, desde teléfonos hasta tabletas y dispositivos plegables.

34 Objetivos del Proyecto

- Implementar diseños adaptativos y responsivos con Jetpack Compose.

- Mostrar cómo crear composables reutilizables para la interfaz de usuario.
- Gestionar múltiples pantallas con navegación integrada.
- Mejorar la experiencia de usuario en dispositivos con diferentes resoluciones.

35 Estructura del Proyecto

- `app/`: Código fuente principal de la aplicación.
- `build.gradle.kts`: Configuración del proyecto con Kotlin DSL.
- `gradle/`: Scripts y configuraciones de Gradle.
- `README.md`: Documentación general y guía para iniciar.

36 Componentes Clave

36.1 MainActivity.kt

Listing 18: MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            ReplyApp()
        }
    }
}
```

Esta clase inicializa la aplicación y establece la función composable raíz `ReplyApp()`.

36.2 ReplyApp.kt

Aquí se configura el diseño adaptable y la navegación entre las diferentes pantallas del cliente de correo.

Listing 19: Ejemplo de navegación y layout adaptativo

```
@Composable
fun ReplyApp() {
    // Configura Scaffold con top bar, navigation drawer y
       contenido adaptativo
    val windowSize = rememberWindowSizeClass()
    when (windowSize.widthSizeClass) {
        WindowWidthSizeClass.Compact -> {
            // Layout para pantallas peque as (m viles)
        }
        WindowWidthSizeClass.Medium -> {
```

```
            // Layout para pantallas medianas (tabletas)
        }
        WindowWidthSizeClass.Expanded -> {
            // Layout para pantallas grandes (PC o plegables)
        }
    }
}
```

36.3 Diseño Adaptativo

El proyecto utiliza la librería `WindowSizeClass` para detectar el tamaño de pantalla y adaptar el diseño UI, mejorando la experiencia del usuario en dispositivos variados.

37 Introducción

La aplicación **Sports** es una app sencilla que muestra noticias deportivas. Este proyecto forma parte del curso *Android Basics with Compose* ofrecido por Google Developer Training.

El objetivo principal de la aplicación es ilustrar el concepto de diseño adaptativo utilizando Jetpack Compose, permitiendo que la interfaz de usuario se ajuste dinámicamente a diferentes tamaños y tipos de pantalla, desde teléfonos hasta tabletas y dispositivos plegables.

38 Objetivos del Proyecto

- Implementar diseños adaptativos y responsivos con Jetpack Compose.
- Mostrar cómo crear composables reutilizables para la interfaz de usuario.
- Gestionar múltiples pantallas con navegación integrada.
- Mejorar la experiencia de usuario en dispositivos con diferentes resoluciones.

39 Estructura del Proyecto

- `app/`: Código fuente principal de la aplicación.
- `build.gradle.kts`: Configuración del proyecto con Kotlin DSL.
- `gradle/`: Scripts y configuraciones de Gradle.
- `README.md`: Documentación general y guía para iniciar.

40 Componentes Clave

40.1 MainActivity.kt

Listing 20: MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            SportsApp()
        }
    }
}
```

Esta clase inicializa la aplicación y establece la función composable raíz `SportsApp()`.

40.2 SportsApp.kt

Aquí se configura el diseño adaptable y la navegación entre las diferentes pantallas de la aplicación.

Listing 21: Ejemplo de navegación y layout adaptativo

```
@Composable
fun SportsApp() {
    // Configura Scaffold con top bar, navigation drawer y
       contenido adaptativo
    val windowSize = rememberWindowSizeClass()
    when (windowSize.widthSizeClass) {
        WindowWidthSizeClass.Compact -> {
            // Layout para pantallas peque as (m viles)
        }
        WindowWidthSizeClass.Medium -> {
            // Layout para pantallas medianas (tabletas)
        }
        WindowWidthSizeClass.Expanded -> {
            // Layout para pantallas grandes (PC o plegables)
        }
    }
}
```

40.3 Diseño Adaptativo

El proyecto utiliza la librería `WindowSizeClass` para detectar el tamaño de pantalla y adaptar el diseño UI, mejorando la experiencia del usuario en dispositivos variados.