Universidad de Granada

Facultad de Ingeniería Informática

Practica 2: Diseño adaptativo en Flutter

UNIVERSIDAD DE GRANADA

DS: Grupo 1.7

Emanuel Giraldo Herrera
Thomas Lang
Timur Sorokin
Alejando Iborra Morán

(2023-2024)

Contents

1 Abstract

En este documento se detalla el proceso de adaptación del ejercicio 3 de la práctica 1 desarrollado en Java al nuevo framework denominado Flutter. La tarea principal consistió en realizar el mantenimiento adaptativo y perfectivo de modo que se ha migrado de un lenguaje a otro y se añadido una interfaz gráfica para permitir la interacción del usuario con el sistema.

Durante el proceso de migración a Dart nos hemos encontrado con las diferencias fundamentales entre los dos lenguajes como es el caso de sintaxis y el manejo de valores NULL.

En cuanto al diseño de la interfaz de usuario (UI), se priorizó la funcionalidad sobre la estética, ya que nuestro objetivo consistia en presentar un prototipo que permitiera mostrar la interacción del usuario con el sistema. De modo que la interfaz se diseñó de manera sencilla y eficiente, centrándose sobretodo en la usabilidad y la claridad de la información presentada.

Finalmente, el resultado obtenido fue una pequeña aplicación que permite al usuario crear personajes seleccionando diferentes opciones y muestra los datos del personaje creado.

2 Introducción

En esta práctica se ha introducido un nuevo framework llamado Flutter desarrollado por Google. Este ofrece una serie de herramientas a los desarrolladores para crear aplicación multiplataforma con una sola base de código, es decir, el desarrollador escribe el código una vez y este puede ser compilado para diferentes plataformas como linux, windows, ios y android.

El concepto central de Flutter se basa en el uso de widgets, los cuales representan los componentes visuales de la aplicación. Estos widgets son altamente personalizables y pueden ser extendidos con facilidad, lo que permite a los desarrolladores crear interfaces de usuario de manera rápida y eficiente para múltiples plataformas. Este enfoque no solo agiliza el proceso de desarrollo, sino que también reduce los costos asociados con el mantenimiento de aplicaciones para diferentes plataformas.

Sin embargo, a pesar de sus beneficios, Flutter también presenta algunas desventajas que deben ser consideradas: - Curva de aprendizaje: Para los desarrolladores nuevos en Flutter, puede haber una curva de aprendizaje significativa, especialmente si no están familiarizados con el paradigma de desarrollo basado en widgets. - Legibilidad del código con widgets anidados: En los casos donde se utilizan múltiples widgets anidados la legibilidad del código disminuy drasticamente dificultando el mantenimiento y la comprensión del código. - Tamaño de la aplicación más grande: Debido a la inclusión del motor de Flutter y otros recursos necesarios en cada aplicación, el tamaño del ejecutable compilado suele ser más grande en comparación con las aplicaciones nativas. - SetState repinta todos los elementos: En Flutter, el uso de la función setState() para actualizar el estado de un widget provoca que todos los elementos se vuelvan a pintar, lo que puede afectar el rendimiento de la aplicación en casos de actualizaciones frecuentes del estado.

A modo de conclusión podemos decir que a pesar de las desventajas mencionadas Flutter sigue siendo una opción relevante en el entorno de desarrollo crossplatform. Además, siendo un proyecto respaldado por Google da cierta garantia y seguridad en cuanto al futuro de este.

2.1 Análisis del problema

A partir de la información expuesta en el guion de la practica se identifican claramente las siguientes dos propuestas:

1. Realizar el mantenimiento adaptativo: Esta fase se realiza la migración del código existente en Java, obtenido en prácticas anteriores, al lenguaje Dart. El objetivo es adaptar el código para que sea compatible con el nuevo framework Flutter.

2. Realizar el mantenimiento perfectivo: En esta etapa se introduce un nuevo componente al sistema que es una interfaz de usuario (UI). Para ello recurimos al uso de los Widgets que nos permiten crear componentes visuales de la aplicación. Se han considerado los aspectos tales como la navegación entre pantallas, la disposición de los elementos de la interfaz y la respuesta a las acciones del usuario.

2.2 Ingeniería inversa

No se qué poner aqui. Hacemos algo de ingenieria inversa en estas práctica?

3 Desarrollo

3.1 Fase 1: Migración a Dart

Para afrontar la migración del código de un lenguaje a otro, hemos dedicado un tiempo para familiarizarnos con la sintaxis de Dart. Gracias a los ejercicios indivuales propuestos tuvimos oportunidad de obtener un poco de experiencia en el manejo de este lenguaje. Dada la simplicidad de este proceso, no consideramos necesario profundizar en detalles específicos.

Posteriormente, hemos procedido a revisar la relación entre las entidades del sistema. En particular, hemos prestado especial atención a las clases Fachada y Menú. Dado que en la practica anterior la interacción con el sistema se llevaba a cabo mediante la terminal era necesario unir los modulos mediante una clase Fachada. No obstante, ahora, puesto que hemos añadido una interfaz grafica (UI) su presencia es dudosa cuanto menos.

Despues de debatirlo, hemos decidido mantener la estructura previa del código intacta. Esta decisión se basa en la utilidad potencial de la clase Menú como herramienta para la depuración y las pruebas en entornos donde no sea necesaria una interfaz gráfica.

3.2 Fase 2: UI

Durante esta fase del proyecto, nos hemos dedicado a revisar la documentación sobre los Widgets en Flutter. Utilizando los recursos disponibles en la página oficial del proyecto, hemos encontrado ejemplos prácticos que han sido de gran ayuda para el desarrollo de esta etapa de la práctica.

Puesto que al tratarse de un proyecto en el campo de la ingeniería software, hemos adoptado un enfoque más practico en cuanto al diseño de la interfaz, priorizando la funcionalidad y claridad sobre consideraciones estéticas. Al final de todo, solo pretendemos mostrar un prototipo que demuestre la interacción entre distintos componentes del sistema.

Por ultimo, consideramos importante mencionar que la clase CreadorDePersonajes juega un papel clave en la aplicación. Esta clase se encarga de manejar la lógica relacionada con la interacción del usuario y la creación de personajes. En otras palabras, se asemeja al concepto de Controlador en el modelo MVC.

3.3 Problemas encontrados

Durante el desarrollo de la aplicación han sido diversos los problemas encontrados. Entre estos podemos nombrar algunos sin mucha relevancia que aún así impedían el correcto funcionamiento de la aplicación y otros que han supuesto un verdadero dolor de cabeza.

- Problema de lectura de los HasMaps:

  Las características de los personajes vienen definidas por su clase y raza y se representan en HasMaps que almacenan el nombre del atributo correspondiente y su valor numérico. Estos atributos y sus valores se encuentran en los archivos .txt de cada clase y raza. La idea es leer estos archivos para generar así los HashMaps con las claves y valores específicos.

  Para ello en un primer momento se había optado por utilizar openRead() seguido de funciones para separar las líneas y posteriormente iterar en cada una para extraer los datos. Sin embargo, el problema es que no se hacía correctamente la iteración de las líneas por problemas con las funciones que manejaban el archivo.

  Debido a que ese enfoque no ha resultado fructífero en ninguna de sus variantes, hemos decido utilizar otra aproximación usando la función readAsLinesSync(), pudiendo acceder a las líneas del archivo directamente y con la misma idea de iterar en cada una de ellas para extraer los datos.

- Errores con la clase Ladrón:

  Con la aplicación aparentemente funcionando de forma correcta, seleccionar la clase Ladrón no devolvía un personaje con dicha clase. Después, de un análisis rápido hemos visto que el problema era utilizar 'ladrón', con la tilde correspondiente, como selectedClass en el boton de la clase Ladrón, mientras que el procesamiento en la función _crearPersonaje() llamaba al LadronBuilder(), solo en el caso de que estuviera seleccionado 'ladron', sin tildar. La solución ha sido tan sencilla como unificar la cadena de caracteres en ambos casos a 'ladron'.

3.4 Widgets utilizados

- MyApp: Se trata de StatelessWidget que define el entry point (main)
- CreadorDePersonajes: Es de tipo StatefulWidget que representa la pantalla principal de la aplicación donde se crea un personaje.
- _CreadorDePersonajes: Se utiliza para representar el estado de la pantalla principal. (raza seleccionada, nombre del personaje...)
- RaceButton: Es un widget sin estado. Encapsula definiciones de estilo para todos los botones de razas (reutilización del código)
- ClassButton: Misma idea que RaceButton. Se utiliza para no repetir el mismo código.
- CharacterDetailsScreen: Es un widget sin estado. Se trata de una vista con los detalles del personaje creado.
- buildLineaAtributo: Este widget sin estado se utiliza para construir una línea de atributo en la pantalla de detalles del personaje.

3.5 Mejoras

4 Ejecución

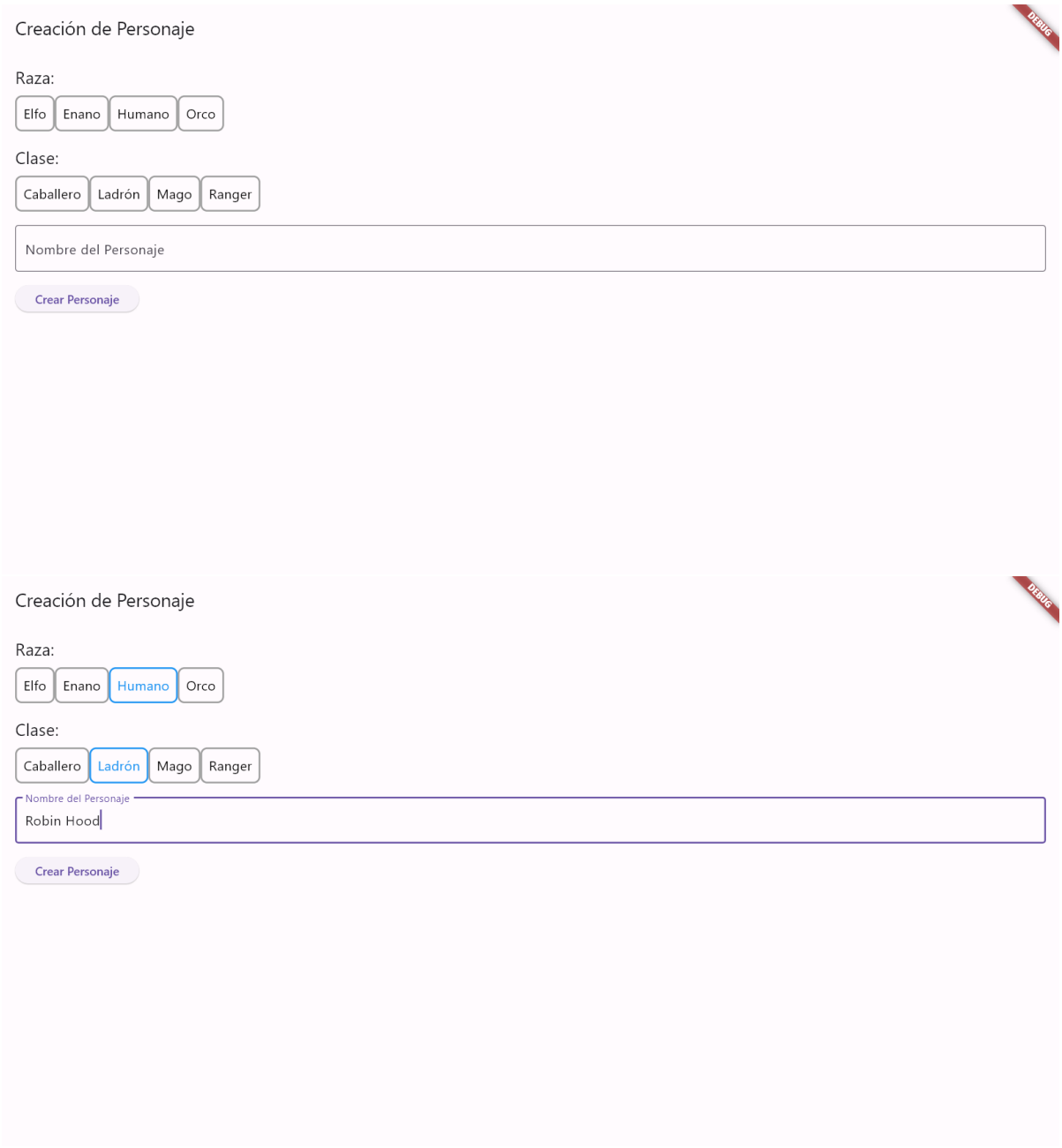

Figure 1: Pantalla de selección de personaje

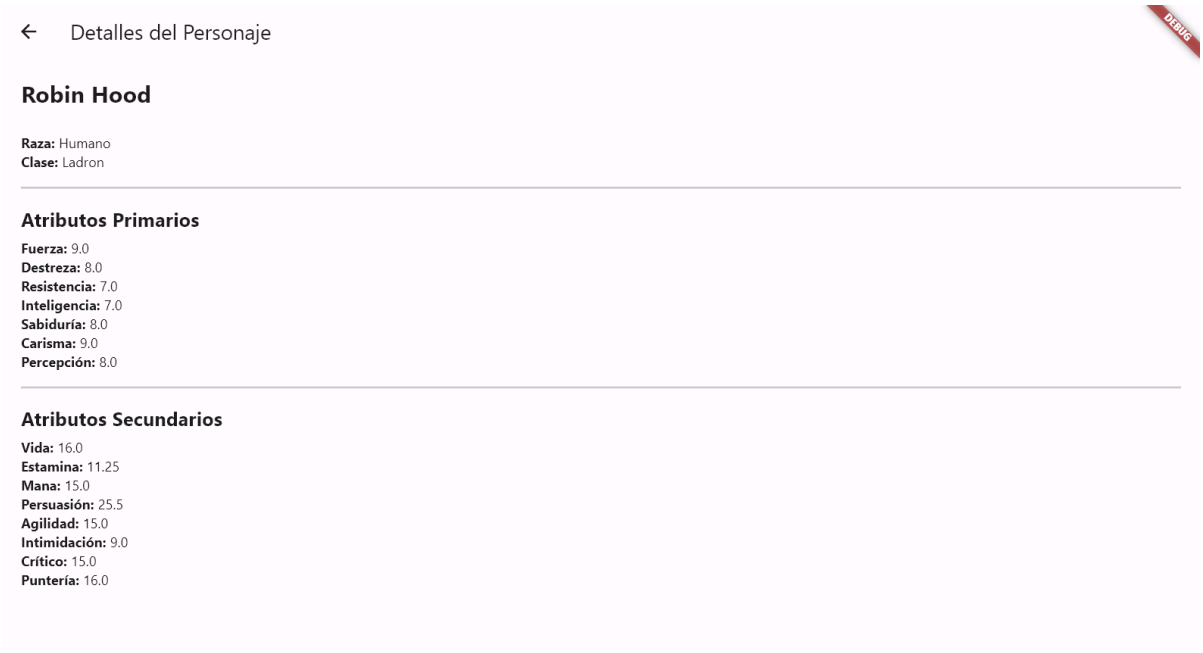

Figure 2: Pantalla de detalle del personaje