**Computer Science Internal Assessment: French Vocabulary Memorization Software**

CRITERION C – DEVELOPMENT:

**Techniques Used:**

1. Defining The ‘Quiz’ Class
2. Using the quizGenerator Function
3. Creating a File Operations System to Track the User’s Progress
4. Using the Map Data Structure to Create a Routing System
5. Using a Random Number Generator to Simulate Battles
6. Using Custom Font Packages and TextStyle Widgets to Improve the User Interface
7. Using a TextEditingController class to receive answers from the user
8. Using Scaffolds to Create Consistent UI
9. Using Futures as Time Delays

*Note: See Appendix B for full code*

**Defining A ‘Quiz’ Class:**

The French Vocabulary Memorization Software represents each test using a ‘Quiz’ object that has parallel lists as properties: one list of questions and list of answers. Each list is assigned to an instance variable of the Quiz class using a parameterized constructor in conjugation with the this keyword.

The Quiz class is defined below:

class Quiz {  
 List<String> questionList;   
 List<String> answersList;

// defining the instance variables of the class  
 Quiz(List questionsList, List answersList){   
 this.questionList = questionsList;  
 this.answersList = answersList;

//using a parameterized constructor and the ‘this’ keyword  
 }  
}

**Using the quizGenerator Function:**

To generate tests, the software uses quizGenerator. quizGenerator randomly generates a Set of ten integers with a for loop. These integers represent the index numbers of question-and-answer pairs stored in the parallel lists of questions and answers. By referencing these index numbers, quizGenerator picks ten questions and answers from the parallel lists, and uses them to create the two lists that are a Quiz object’s properties. These two lists are subsequently inserted into a parameterized constructor to create a new instance of the Quiz class. This method of test creation ensures questions are not repeated, as elements cannot be duplicated in a Set.

quizGenerator works as follows:

Quiz quizGenerator (questions, answers) {  
 final Set<int> indexS = Set(); //Creating a Set   
 for (int i = 0; i <= 150; i++){  
 indexS.add(Random().nextInt(noQuestions-1));

//Populating the set with random integers in the range 0 to (noQuestions-1)  
 }  
 List<String> testQ = [], testA = [];

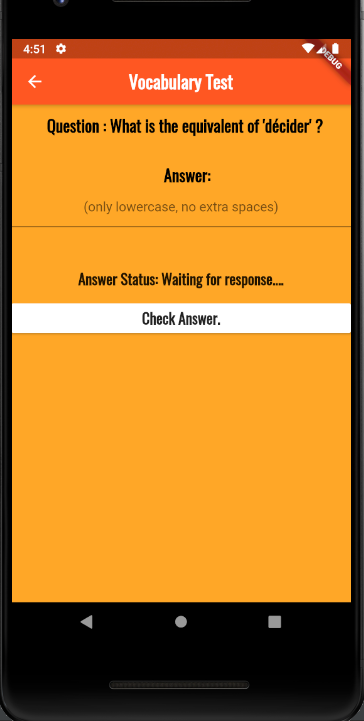
//Initializing the properties of the Quiz object  
 List<int> indexes = indexS.toList();

//Converting the Set to a list so that the index numbers in it can be accessed easily  
 for (int i = 0; i <= 10; i++){  
 var element = indexes[i];  
 testQ.add(questions[element]);   
 testA.add(answers[element]);

//Adding the question-and-answer pairs at each randomly-generated index number to the two lists that are a Quiz object’s properties   
 }  
 var newTest = Quiz(testQ, testA);

//Passing the two lists into a parameterized constructor  
 return newTest; //Returning the Quiz object  
}

**Figure 1: Result of Test Generation**



**Creating A File Operations System to Track the User’s Progress:**

The code for this was derived from here[[1]](#footnote-1).

To track the user’s progress, their score on the last test is compared with their score on the current test. The system works by writing the user’s score to a text file, and later retrieving it for comparison purposes. The system first obtains the path to an available directory on the device where user-generated data can be stored, and creates a text file named lastScore.txt there. Depending on which function is called, the system either takes the user’s current score as a parameter and writes it to the text file, or reads the contents of the text file and stores it in a variable. In the event that lastScore.txt is empty, the read function returns an empty string to avoid raising an exception.

The system benefits from using static futures, which save memory space by only initializing variables when they are used in the software. Additionally, the use of asynchronous functions allows the software to multitask.

The code for the file operations system is as follows:

import 'package:path\_provider/path\_provider.dart';  
import 'dart:io';  
  
class ScoreOperations {   
 static Future<String> get *getScorePath* async {  
 final scorePath = await getApplicationDocumentsDirectory();  
 return scorePath.path;

//getting the path of an available directory  
 }  
  
 static Future<File> get *getScore* async {  
 final scoreP = await *getScorePath*;  
 return File('$scoreP/lastScore.txt');

//creating a local text file in the available directory  
 }  
  
 static Future<File> *saveScore*(String data) async {  
 final scoreFile = await *getScore*;  
 return scoreFile.writeAsString(data);

//taking in the user’s current score as a parameter and writing it to the text file  
 }  
  
 static Future<String> *readScore*() async {  
 try {  
 final scoreFile = await *getScore*;  
 String pastScore = await scoreFile.readAsString();  
 return pastScore;

//reading the user’s last score from the text file  
 } catch (e) {  
 return "";  
 }  
 }  
}

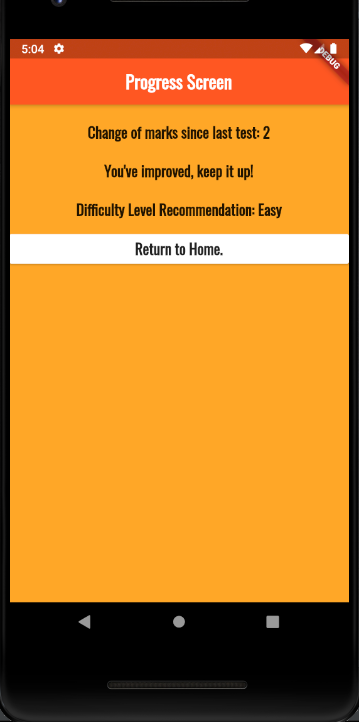
An example of how *ScoreOperations* is used:

if (10 == counter) {  
ScoreOperations.*readScore*().then((score) {   
 setState(() {  
 lastScoreM = int.*parse*(score);  
 });  
});

//reading the user’s last score from the text file and storing it in a variable for comparison purposes  
Future.delayed(const Duration(milliseconds: 2300), () {  
 ScoreOperations.*saveScore*(noCorrect.toString());  
});  
}

//writing the user’s current score to the text file after a delay so that it becomes the new ‘last score’

**Figure 2: The Progress Page**



**Using the Map Data Structure to Create a Routing System:**

A routing system was created in *main.dart* to easily direct the user between the different pages in the software. This routing system is accomplished using Dart’s map data structure. A map is a collection of ‘key-value’ pairs, where each ‘key’ (the route name) maps to exactly one ‘value’ (a build function that returns the widget for the appropriate page). A context variable specifying the position of said widget in the widget tree of the software is also passed in as an argument to this build function. This routing system facilitates abstraction in the software, as it is not necessary to instantiate a particular route every time it is used.

From *main.dart*:

import 'package:flutter/material.dart';  
import 'package:french\_learning\_ia/pages/battlemode.dart';  
import 'package:french\_learning\_ia/pages/bmeasy.dart';  
import 'package:french\_learning\_ia/pages/bmhard.dart';  
import 'package:french\_learning\_ia/pages/bmmedium.dart';  
import 'package:french\_learning\_ia/pages/easytest.dart';  
import 'package:french\_learning\_ia/pages/esa.dart';  
import 'package:french\_learning\_ia/pages/eta.dart';  
import 'package:french\_learning\_ia/pages/hardtest.dart';  
import 'package:french\_learning\_ia/pages/home.dart';  
import 'package:french\_learning\_ia/pages/hpa.dart';  
import 'package:french\_learning\_ia/pages/hsa.dart';  
import 'package:french\_learning\_ia/pages/hta.dart';  
import 'package:french\_learning\_ia/pages/pa.dart';  
import 'package:french\_learning\_ia/pages/pea.dart';  
import 'package:french\_learning\_ia/pages/progress.dart';  
import 'package:french\_learning\_ia/pages/sa.dart';  
import 'package:french\_learning\_ia/pages/scoreE.dart';  
import 'package:french\_learning\_ia/pages/scoreH.dart';  
import 'package:french\_learning\_ia/pages/test.dart';  
import 'package:french\_learning\_ia/pages/score.dart';  
import 'package:french\_learning\_ia/pages/progressE.dart';  
import 'package:french\_learning\_ia/pages/progressH.dart';  
import 'package:french\_learning\_ia/pages/testA.dart';

// Imports all the files in the software to access their widgets

void main() => runSoftware(MaterialSoftware(  
 initialRoute: '/', // Defines the home page/initial route of the software   
 routes: { // Creates a map of routes called ‘routes’  
 '/': (context) => Home(),  
 '/test': (context) => Test(),  
 '/testA': (context) => TestA(),  
 '/score': (context) => Score(),  
 '/progress': (context) => Progress(),  
 '/ht': (context) => HardTest(),  
 '/hta': (context) => HardTestA(),  
 '/et': (context) => EasyTest(),  
 '/eta': (context) => EasyTestA(),  
 '/es': (context) => ScoreE(),  
 '/hs': (context) => ScoreH(),  
 '/ep': (context) => ProgressE(),  
 '/hp': (context) => ProgressH(),  
 '/esa': (context) => ScoreEA(),  
 '/epa': (context) => ProgressEA(),  
 '/hpa': (context) => ProgressHA(),  
 '/ta': (context) => TestA(),  
 '/hsa': (context) => ScoreHA(),  
 '/sa': (context) => ScoreA(),  
 '/pa': (context) => ProgressA(),  
 '/bm': (context) => BattleMode(),  
 '/bme': (context) => EasyBM(),  
 '/bmm': (context) => MediumBM(),  
 '/bmh': (context) => HardBM(),

// List of key-value pairs, where ‘keys’ are separated from ‘values’ by a colon  
 }  
));

Using this map, if the user needs to be directed to a particular page, only the following code is required:

Navigator.*pushReplacementNamed*(context, '/score');

//passing in a context variable and a ‘key’ (the route name)

**Using a random number generator to simulate battles:**

To simulate the software that the user will play against in ‘Battle Mode’, a random number generator first generates a number between 1 and 20. This number is then checked to see if it falls in a certain ‘correct’ range determined by the difficulty level the user has selected. If the number falls in the ‘correct’ range, then the software is said to have answered the question correctly. For instance, if the ‘correct’ range for a Software is from 1 to 14, the software, will, on average, be 70% correct, corresponding to a medium difficulty level battle. This method of setting difficulty levels means they can be tweaked exactly to the user’s tastes, which improves their experience with the software.

Initializing the variables involved:

var issoftwareCorrect;  
var accuracyDeterminer = new Random ();

//creating a new instance of the random class  
var accuracyNumber = accuracyDeterminer.nextInt(20);

//generating a random integer in the range of 1 to 20  
var howGood = 9; //variable defining how good the software is. The value of this variable changes with the difficulty level of the battle.  
var isCorrect = false;  
var answerMessage = "Waiting for response....";  
var softwareAnswer = "Waiting for answer...";  
var wrong;

if (accuracyNumber >= 0 && accuracyNumber <= howGood) {

//using howGood to check if the random number falls in the ‘correct’ range  
 setState(() {  
 issoftwareCorrect = true; //the software is correct  
 });  
 }  
 if (accuracyNumber >= (howGood+1) && accuracyNumber <= 20) {  
 setState(() {  
 issoftwareCorrect = false; //the software is incorrect since the random number generated is out of the ‘correct’ range  
 });  
 }

if (issoftwareCorrect == true) {  
 setState(() {  
 nosoftwareCorrect = nosoftwareCorrect + 1;   
 displaynosoftwareCorrect = nosoftwareCorrect.toString();

//the software’s score was incremented by 1 and the new value displayed  
 softwareAnswer = answers[element];

//setting the software’s answer to be the correct answer  
 });  
 }  
 else if (issoftwareCorrect == false) {  
 setState(() {  
 do {  
 wrong = question.nextInt(noQuestions);  
 } while (wrong == element);  
 softwareAnswer = answers[wrong];

//displaying the software’s answer as some randomly-selected incorrect answer  
 });  
 }   
 },  
 ),  
 ),  
 ),  
 ),

**Figure 3: The Battle Mode Interface**

**A screenshot of a phone

Description automatically generated with medium confidence**

**Using Custom Font Packages and TextStyle Widgets to Improve the User Interface:**

The software uses a custom font package named ‘Oswald’ as a dependency through the *pubspec.yaml* file. ‘Oswald’ is added as a dependency by downloading accessing the font’s .ttf (TrueType font) files, which describe the glyph outlines of the font. ‘Oswald’ is then used in conjunction with *TextStyle* widgets to appropriately style the text displayed on screen.

From the *pubspec.yaml* file:

fonts:   
 - family: Oswald   
 # specifying dependency type and font name

fonts:  
 - asset: fonts/Oswald-Regular.ttf

# accessing the .ttf file for the regular version of the Oswald Font

Example usage of Oswald in a TextStyle widget:

Padding(  
 padding: const EdgeInsets.fromLTRB(15.0, 0.0, 0.0, 0.0),

// adding spacing around the text widget  
 child: Text(  
 'Would you like to take another test?',  
 style: TextStyle(

//adding a TextStyle widget as a style property of the Text widget  
 fontFamily: 'Oswald', //specifying the font name  
 fontSize: 22.0, //specifying its size  
 fontWeight: FontWeight.*bold*, //bolding the font  
 ),  
 ),  
),

**Figure 4: Example of Oswald Being Used on Screen**

A picture containing graphical user interface

Description automatically generated

**Using a TextEditingController class to receive answers from the user:**

To receive the user’s answer to a question, an instance of the TextEditingController class is created. This controller acts as a callback function that updates the value of the variable associated with the content of the TextField widget when the TextField is submitted.

Creating a new instance of the TextEditingController class:

var myController = TextEditingController();

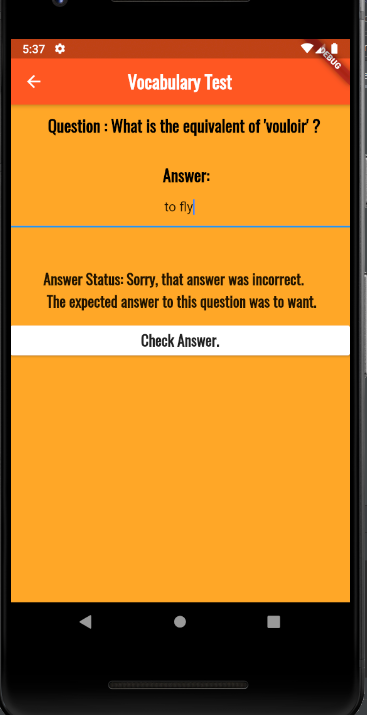
Using the controller in a TextField widget:

TextField(  
 controller: myController, //assigning the instance to be the controller for the TextField widget  
 textAlign: TextAlign.center,  
 decoration: InputDecoration(  
 contentPadding: EdgeInsets.*zero*,  
 hintText: "(only lowercase, no extra spaces)", //telling the user how they should enter their answer  
 ),  
 onSubmitted: (answer){

//indicating that the content of the TextField widget will be stored in the variable ‘answer’ when the widget is submitted  
 },  
),

answer = myController.text; //storing the content of the TextField widget in ‘answer’ using the controller. This is done as the ‘Submit’ Button linked to the TextField widget is pressed.

**Figure 5: The Software Receiving The User’s Answers:**



**Using Scaffolds to Create Consistent UI:**

To ensure the software has a consistent look, I used a build function returning a Scaffold widget to create the general layout for an empty page. I then used this Scaffold widget as the starting point for the layouts of all the other pages in the software.

The scaffold widget:

Widget build(BuildContext context) {

//passing in information on the Scaffold widget’s relation to other widgets in the widget tree of the software. This is achieved through a context variable  
 return Scaffold( //returning a Scaffold widget  
 backgroundColor: Colors.*orange*[400],

//specifying a background color for the page  
 appBar: AppBar( //creating a bar to display the title of the page  
 title: Text(  
 '', // empty title  
 style: TextStyle( //font styling  
 fontFamily: 'Oswald',  
 fontSize: 22.0,  
 fontWeight: FontWeight.*bold*,   
 ),  
 ),  
 leading: IconButton( //creating a back button  
 onPressed: () {  
 Navigator.*pushReplacementNamed*(context, '/bm');  
 },  
 icon: Icon(Icons.*arrow\_back*),   
 ),  
 backgroundColor: Colors.*deepOrange*,   
 centerTitle: true,

// centring the Appbar’s text and giving the appBar a color  
 ),  
 body: SingleChildScrollView( // makes the page scrollable  
 child: Column(  
 children: <Widget>[

// adds a column widget that displays its children widgets in a vertical array on the page   
 ],  
 ),  
 ),  
 );  
 }  
}

**Figure 6: the Scaffold Widget When Being Displayed**

Shape, square

Description automatically generated

**Using Futures as Time Delays:**

After the user submits their answer to a question, there is a short time delay before the software moves on to the next question. This is to give the user a few seconds to see whether their answer is correct, and if not, to see the correct answer to the question. This delay is implemented by creating an empty future that waits for some specified, constant length of time, before executing the code inside the future statement. Only then is the future completed.

For example:

Future.delayed(const Duration(milliseconds: 2500), () {

//creating an empty future and specifying the time delay   
 setState(() {  
 word = currentTest.questionList[counter];   
 answerMessage = "Waiting for response....";  
 });

// executing the code inside the Future statement  
});

**Word count: 996**

**Bibliography:**

James. “File Operations in Flutter – Read and Write Files – Easiest Example.” MOBILE PROGRAMMING, 26 Dec. 2018, www.coderzheaven.com/2018/12/26/file-operations-in-flutter-read-and-write-files-easiest-example.

1. James. “File Operations in Flutter – Read and Write Files – Easiest Example.” MOBILE PROGRAMMING, 26 Dec. 2018, www.coderzheaven.com/2018/12/26/file-operations-in-flutter-read-and-write-files-easiest-example. [↑](#footnote-ref-1)