PROJET 5: CountOnMe

INTRODUCTION

Bonjour, je suis Duc Lucas.

Je vais vous montrer mon projet CountOnMe, c'est une application calculatrice pour iPhone. Ce projet est lié à mes études sur OpenClassrooms. Et il y aura 4 étapes dans ma présentataion :

1) Démonstration de l'application

2) Présentation de l'architecture MVC

3) Présentation de l'interface

4) Présentation du code

1) DEMONSTRATION DE L'APPLICATION (voir simulator)

On va voir sur le simulateur. Et je vous montre les déroulements d'un calcul, par exemple:

- les operations simples (3 + 2 = 5)
- les operations prioritaires (2 + 2 x 2 = 6)
- le nombre en décimal (100 ÷ 3 = 333.333)
- la puissance (10000000000 x 10000000000 = 1e20)

Ensuite, le message d'alerte s'affiche pour:

- les opérateurs à la suite
- la division par zéro

2) PRESENTATION DE L'ARCHITECTURE MVC (page 4)

- **View** représente la structure et l'apparence de ce que voit l'utilisateur, elle reçoit l'interaction de l'utilisateur avec la vue. Et elle doit transmettre

la gestion au ViewController.

- **ViewController** représente le contrôle et la connexion entre la View et le Model, elle reçoit l'entrée utilisateur et la valide. Et elle doit appliquer les propriétés et les commandes du Model pour les envoyer à la View.
- **Model** représente le réceptacle des inputs et des outputs du ViewController. Il doit contenir et définir des propriétés.
- **Test Unitaire** représente la procédure de vérification du Model, je vous montrerai ça dans la présentation du code.

En résumé:

MVC permet de faciliter la séparation en trois parties View, Controller et Model. En raison de la séparation des responsabilités, le développement ou la modification futur est plus simple.

3) PRESENTATION DE L'INTERFACE (page 5)

On remarque que l'interface n'est pas terminée, car le développeur n'a pas fait les contraintes pour la responsive, manque des touches pour multiplication, division et reset.
Donc on refait l'interface, on utilise des StackView pour les touches. On ajoute et supprime les contraintes. Et on vérifie la responsive de iPhone SE à iPhone 11

4) PRESENTATION DU CODE (page 6)

D'abord on découpe MVC du projet pour identifier les codes qui appartiennent au Model. Et on les déplace dans le Model nommé Calculator.swift
Dans la class Calculator, on déclare les variables:

- var calculaString -> pour afficher dans le textview de la View
- var elements -> pour créer les nombres
- les 4 variables permettent de vérifier le calcul
- var isDevideByZero -> pour la division par Zero
- var isPossibleToStartWithNumber -> pour commencer le nombre et non opérateur

On utilise les Outputs

- var alertMessage
- var calculText

Dans ce code qu'on a écrit, on a relié à une fonction. Car elles ont la meme signature.

Rappel: *Une variable réactive, c'est une variable de type closure. Elle possède donc un type de fonction. L'idée ici, c'est relier cette variable à un éléments du controller. Ce qu'on appelle Binder.*
Une fois que ma variable réactive est connectée, dès qu'on lui assignera une valeur dans le model, l'élément du controller auquel elle est reliée s'actualisera automatiquement.

On ajoute les fonctions pour :

- les nombres
- les opérateurs
- le message d'alerte pour la division par zéro
- la suppression avec la touche AC
- le résultat final avec la touche égale

Ensuite on appelle ces fonctions au controller

On teste le bon fonctionnement des propriétés et des commandes (page 15)

1 Pour créer des tests, il faut rajouter une target

2 Ensuite, il faut créer un fichier de test dans lequel on crée une sous-classe de XCTestCase

3 La méthode setUp est rappelée avant chaque test. Elle permet de faire une initialisation

4 On écrit les noms des test en BDD (Behavior Design Development) et on sépare le nom en trois partie: Given When et Then

5 On ajoute la fonction XCTAssert qui permet d'évaluer un booléen s'il vaut true, le test réussit et inversement.

Les Test Unitaires vous permettent de:

- prouver que votre code fonctionne

- prévenir les bugs

- créer une documentation vivante de votre code

- être professionnel

L'application est testée a 100% dans le model (page 16)

Le code coverage permet de visualiser les zones de code qui ne sont pas couvertes par un test. Donc il faut chercher à atteindre les 100% de couvertures à tout prix.

Ma présentation terminée, avez vous des questions?

Rappels:

AppDelegate: c'est tout premier fichier lancé lorsque l'application démarre. Et toute premiere fonction de l'appDelegate, c'est elle là qui est appelé lors du lancement de l'application.

ScèneDelegate a été introduit avec iOS 13, donc il est tout récent. Apple a changé son système de fenêtre pour faire simple, il utilise maintenant des scènes. Et c'est ce fichier qui gère les scènes de votre application. Par exemple on quitte l'application ou on ferme un écran. En gros, la scène delegate qui gère tout ca maintenant.

TDD = Test Driven Development -> est une technique de développement très puissante dans laquelle on écrit les est avant le code.
Concrètement on écrit un test qui échoue puis on écrit le code correspondant, ce qui répare le test.
Les intérêts du TDD sont:
- travailler à ce que l'on fait
- être concentré
- la validation automatique du code
- ne pas oublier les tests

La méthode Red Green et Refactor se compose de 3 étapes:
- Red on écrit un test qui échoue
- Green on résout le test en écrivant le code correspondant

- **Refactor on refactorise le code ET les tests**