Count On Me

Présentation de modification d'application

Vincent Santos

Compétences évaluées :

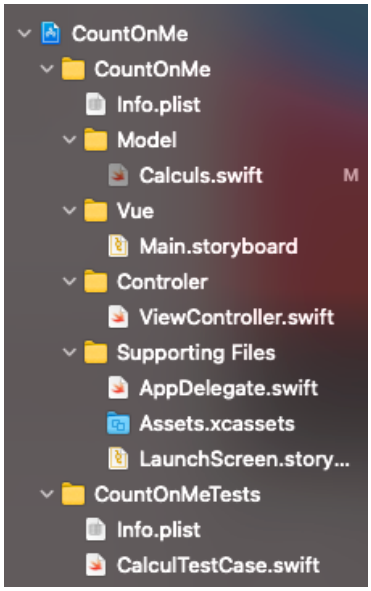
- Mise en place du modèle MVC sur l'application
- Mise en place de toutes les contraintes liées à tout iPhone en mode portrait
- Ajout des fonctions manquantes
- Mise en place des tests unitaires de la partie model

Contraintes techniques :

- En ligne sur gitHub
- Ecrit en anglais
- Respecter le MVC
- Tests Unitaires
- iOS 11 et swift 4 minimum

Mise aux normes MVC :

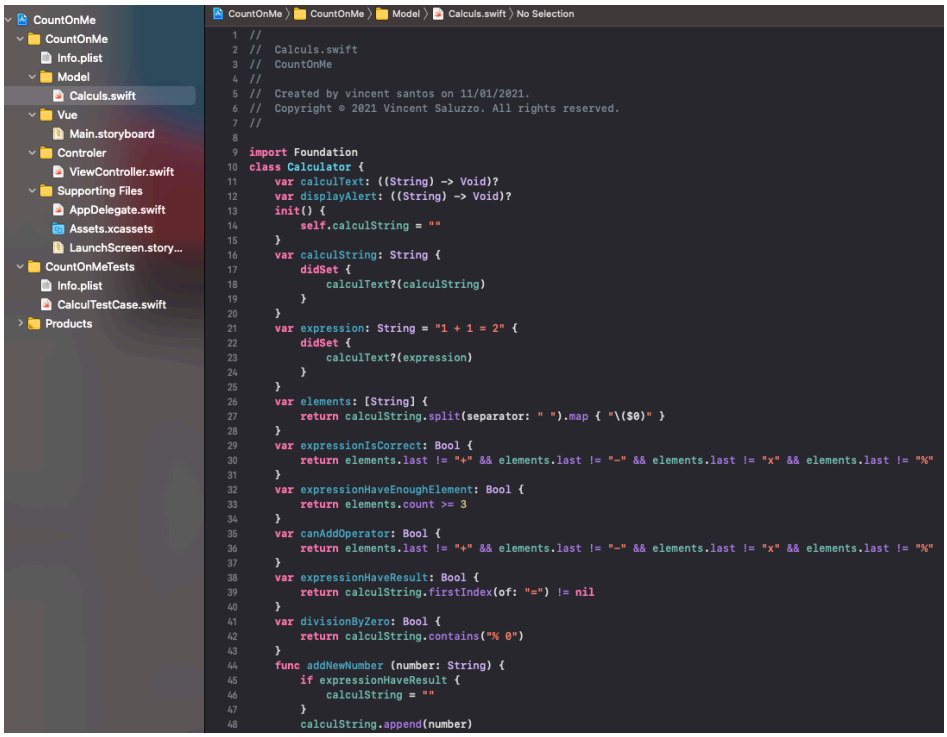
J'ai commencé par modifier l'arborescence en MVC pour plus de clarté.



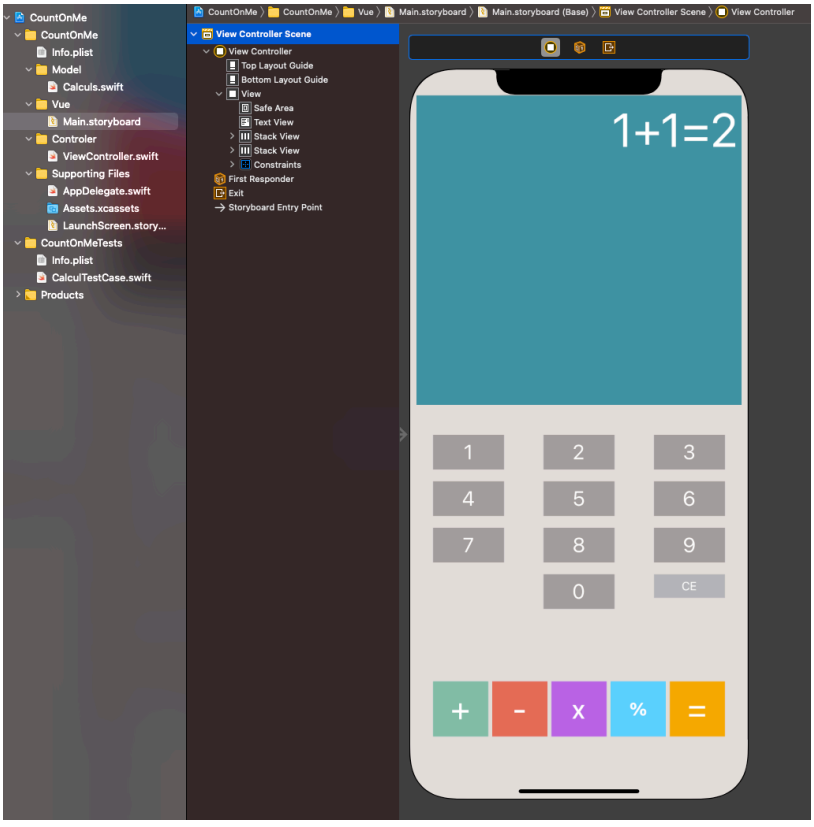
Après avoir créé la partie Controller, pour contenir toute la partie de liaison avec l'utilisateur de l'application: ViewController.swift

```swift
//
//  ViewController.swift
//  SimpleCalc
//
//  Created by Vincent Saluzzo on 29/03/2019.
//  Copyright © 2019 Vincent Saluzzo. All rights reserved.
//

import UIKit

class ViewController: UIViewController {
    @IBOutlet weak var textView: UITextView!
    @IBOutlet var numberButtons: [UIButton]!
    var calcul = Calculator()
    // View Life cycles
    override func viewDidLoad() {
        super.viewDidLoad()
        bind()
    }
    // View actions
    @IBAction func tappedNumberButton(_ sender: UIButton) {
            guard let numberText = sender.title(for: .normal) else {
                return
            }
            calcul.addNewNumber(number: numberText)
        }
    @IBAction func tappedAdditionButton(_ sender: UIButton) {
        calcul.addOperator(operateur: "+")
    }
    @IBAction func tappedSubstractionButton(_ sender: UIButton) {
        calcul.addOperator(operateur: "-")
    }
    @IBAction func tappedMultiplicationButton(_ sender: UIButton) {
        calcul.addOperator(operateur: "x")
    }
    @IBAction func tappedDiviseButton(_ sender: UIButton) {
        calcul.addOperator(operateur: "%")
    }
    @IBAction func tappedEqualButton(_ sender: UIButton) {
        calcul.calculs()
    }
    @IBAction func tappedResetButton(_ sender: UIButton) {
        calcul.clean()
    }
    func bind() {
            calcul.calculText = { [ weak self ] text in
                self?.textView.text = text
            }
            calcul.displayAlert = displayAlert
        }
    func displayAlert(message: String) {
        let alertController = UIAlertController(title: "Attention", message: message, preferredStyle: .alert)
        alertController.addAction(UIAlertAction(title: "ok", style: .cancel, handler: nil))
        present(alertController, animated: true)
    }
}
```

J'ai ensuite créé la partie Model, où j'ai créé le fichier Calculs.swift à l'intérieur, le fichier comme dans toute partie model en Swift, contient toute la partie algorithmique de l'application.



Pour finir, la partie Vue est là pour héberger le Storyboard



Ajout des fonctions de multiplication, de division ainsi qu'une remise à zéro :

Pour ce faire j'ai ajouté 2 boutons dans le Storyboard comme nous pouvons le voir dans l'image précédente.

Je les ai reliés ensuite au controller via 2 IBAction dans le controller.

```
@IBAction func tappedMultiplicationButton(_ sender: UIButton) {
    calcul.addOperator(operateur: "x")
}
@IBAction func tappedDiviseButton(_ sender: UIButton) {
    calcul.addOperator(operateur: "%")
}
```

Ensuite dans le Model , j'ai ajouté le type d'opération à l'intérieur de la méthode de calcul.

```
func calculate(left: Double, right: Double, operand: String) -> Double {

    let result: Double
    switch operand {
    case "+": result = left + right
    case "-": result = left - right
    case "%": result = left / right
    case "x": result = left * right
    default: return 0.0
    }
    return result
}
```

J'ai ajouté également un système de priorité dans les calculs pour la multiplication et la division.

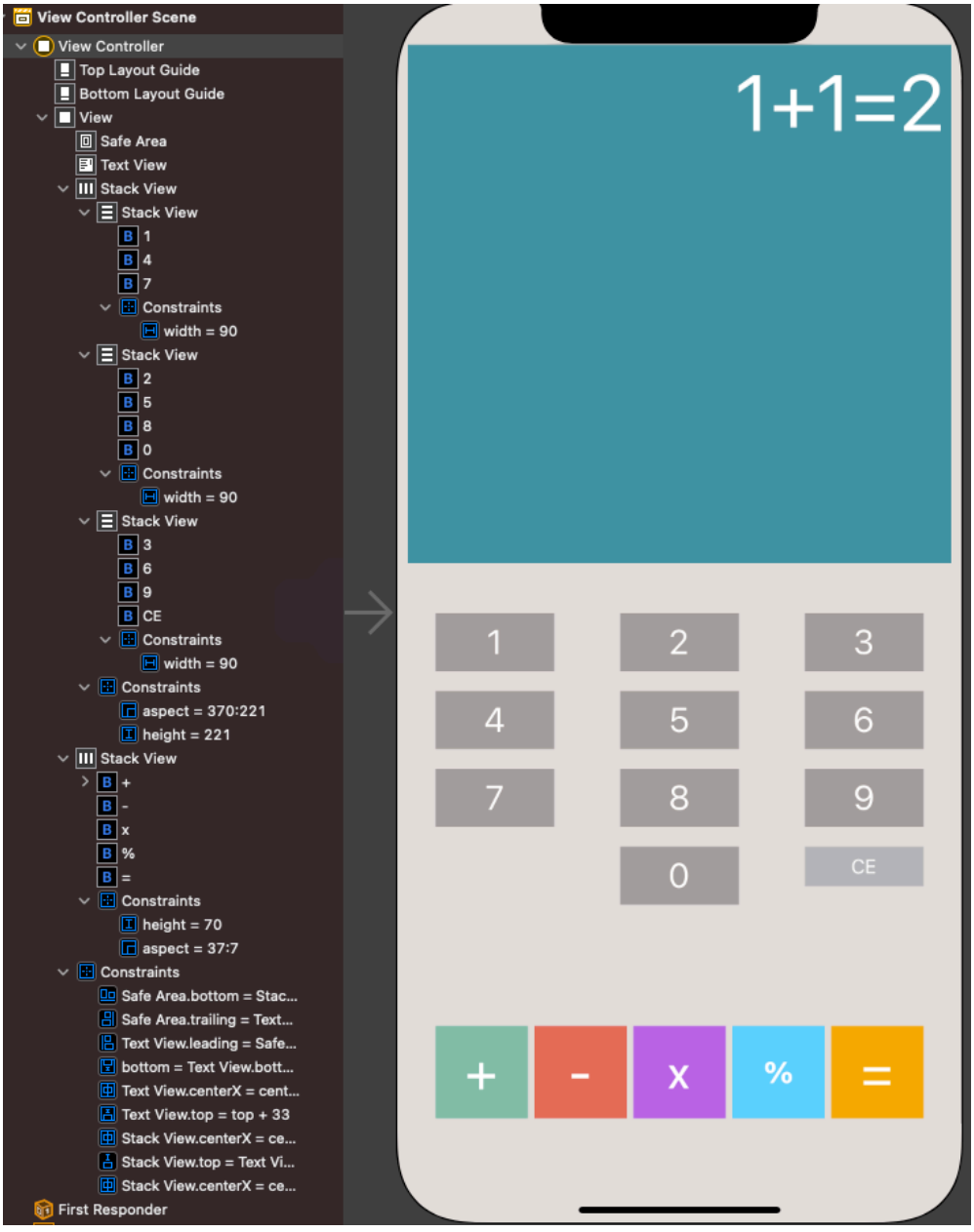
```
var operationsToReduce = elements
while operationsToReduce.count > 1 {
    guard var left = Double(operationsToReduce[0]) else { return }
    var operand = operationsToReduce[1]
    guard var right = Double(operationsToReduce[2]) else { return }
    let result: Double
    var operandIndex = 1
    if let index = operationsToReduce.firstIndex(where: { $0 == "x" || $0 == "%" }) {
        operandIndex = index
        if let leftunwrapp = Double(operationsToReduce[index - 1]) { left = leftunwrapp }
        operand = operationsToReduce[index]
        if let rightUnwrapp = Double(operationsToReduce[index + 1]) { right = rightUnwrapp }
    }
    result = calculate(left: Double(left), right: Double(right), operand: operand)

    for _ in 1...3 {
        operationsToReduce.remove(at: operandIndex - 1)
    }
    operationsToReduce.insert(formatResult(result: Double(result)), at: operandIndex - 1 )
}
guard let finalResult = operationsToReduce.first else { return }
calculString.append(" = \(finalResult)")
```

En ce qui concerne la remise à zéro, j'ai ajouté un bouton « CE » dans le storyboard relié au controller via une IBAction qui appèle la méthode « clean »

```swift
@IBAction func tappedResetButton(_ sender: UIButton) {
    calcul.clean()
}
```

```swift
func clean() {
    calculString.removeAll()
    calculText?("0")
}
```

Contraintes pour tout iPhone en mode portrait :



Mise en place des tests unitaires de la partie model :

J'ai mis en place les 7 principaux tests du model :

- Test d'erreur si pas d'opérateur
- Test d'erreur si nombre manquant
- Test d'erreur si plusieurs opérateurs à la suite
- Test du bon résultat d'une opération simple
- Test de vérification de la priorité des opérations
- Test du bon résultat d'un calcul complexe à plusieurs opérateurs
- Test d'erreur de la division par zéro

```swift
import XCTest
@testable import CountOnMe
class CalculTestCase: XCTestCase {
    // Nous verrifions que si pas d'operateur ont retourne l'alerte
    func testGivenCompleteOperation_WhenNotOperator_ThenReturnErrorString() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "")
        calculator.addNewNumber(number: "2")
        var stringError = ""
        let expectation = self.expectation(description: "operation")
        calculator.displayAlert = { result in
            stringError = result
            expectation.fulfill()
        }
        calculator.calculs()
        waitForExpectations(timeout: 5, handler: nil)
        XCTAssertEqual(stringError, "valeurs manquantes")
    }
    // Nous verrifions que si un nombre de l'operation est manquant ont retourne l'alerte
    func testGivenNumberForget_WhenElementsNumberMissing_ThenReturnErrorString() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "1")
        calculator.addOperator(operateur: "+")
        calculator.addNewNumber(number: "")
        var stringError = ""
        let expectation = self.expectation(description: "number")
        calculator.displayAlert = { result in
            stringError = result
            expectation.fulfill()
        }
        calculator.calculs()
        waitForExpectations(timeout: 5, handler: nil)
        XCTAssertEqual(stringError, "Entrez une valeur correcte")
    }
    // Nous verrifions qu'il y est pas plusieurs opérateurs, sinon ont retourne l'alerte
    func testGivenAddTwoOperators_WhenTwoOperatorDetected_ThenReturnErrorString() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "1")
        calculator.addOperator(operateur: "+")
        calculator.addOperator(operateur: "+")
        var stringError = ""
        let expectation = self.expectation(description: "operators")
        calculator.displayAlert = { result in
            stringError = result
            expectation.fulfill()
        }
        calculator.calculs()
        waitForExpectations(timeout: 5, handler: nil)
        XCTAssertEqual(stringError, "Entrez une valeur correcte")
    }
    // Nous verrifions que nous avons bien le bon résultat a l'opération simple
    func testGivenExpressionHaveResult_WhenExpressionHaveResult_ThenReturnOperation() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "3")
        calculator.addOperator(operateur: "+")
        calculator.addNewNumber(number: "3")
        calculator.calculs()
        XCTAssert(calculator.calculString == "3 + 3 = 6")
    }
    // Nous verrifions que la priorité soit respecter
    func testGivenOrderOfOperations_WhenElementsContainSomething_ThenElementsFollowsOrderOfOperations() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "+")
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "x")
        calculator.addNewNumber(number: "4")
        calculator.calculs()
        XCTAssert(calculator.calculString == "2 + 2 x 4 = 10")
    }
    // Nous verrifions un calcul long
    func testGivenLongOperations_WhenElementsContainSomething_ThenElementsFollowsOrderOfOperations() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "+")
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "x")
        calculator.addNewNumber(number: "4")
        calculator.addOperator(operateur: "+")
        calculator.addNewNumber(number: "4")
        calculator.addOperator(operateur: "%")
        calculator.addNewNumber(number: "2")
        calculator.calculs()
        XCTAssert(calculator.calculString == "2 + 2 x 4 + 4 % 2 = 12")
    }
    // Nous verrifions que nous ne pouvons pas diviser par zéro
    func testGivenDivisionByZero_WhenDivisionByZero_ThenReturnErrorString() {
        let calculator = Calculator()
        calculator.addNewNumber(number: "2")
        calculator.addOperator(operateur: "%")
        calculator.addNewNumber(number: "0")
        var stringError = ""
        let expectation = self.expectation(description: "division")
        calculator.displayAlert = { result in
            stringError = result
            expectation.fulfill()
        }
        calculator.calculs()
        waitForExpectations(timeout: 5, handler: nil)
        XCTAssertEqual(stringError, "Division par zero imposible")
    }
}
```