Link: https://github.com/atalantes-hh/Projet_CountOnMe

Cahier des charges du projet CountOnMe

❖ Contexte

Le manager Juan a besoin d'aide pour finaliser le projet d'un ancien développeur aujourd'hui parti.
Ce développeur était en charge du développement d'une application pour un client.
Juan se rend compte que l'application n'est pas finalisée et le client attend.

❖ Etat du projet

Le développeur travaillait sur une calculatrice.
Le design de l'application est assez basique comme l'avoue Juan.

Les premiers constats sont les suivants :

- ▶ Le design n'est pas finalisé et non conforme au responsive
- ▶ L'architecture du projet ne respecte pas les bonnes pratiques de développement (pas de MVC)
- ▶ L'ensemble de l'application n'est pas testé
- ▶ Seul la soustraction et l'addition ont été faits, il manque la division et la multiplication

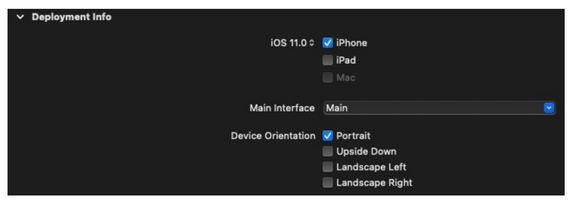
Le projet est disponible sur GitHub

Constats annexes :

- ▶ D'autres fonctions essentielles sont manquantes comme la remise à zéro
- ▶ L'application actuelle est sujette au crash

❖ Contraintes techniques

- ▶ Le code doit être en Anglais et commenté
  - ✓ Condition Respectée
- ▶ Il est disponible sur toutes les tailles d'iPhone en mode portrait
  - ✓ Disponible de l'iPhone 4s au 12 Pro Max
- ▶ Écrit en Swift 4 Minimum
  - ✓ Utilisation de la version Swift 5.3
- ▶ Pas d'erreurs ou de Warning
  - ✓ Condition Respectée
- ▶ Respect du MVC
  - ✓ Condition Respectée (sera détaillé dans la partie Architecture)
- ▶ Être fonctionnel sous Swift 11 et ultérieur
  - ✓ Dans General / Deployment Info cela a été spécifié :



- ▶ Accompagné de tests unitaires
  - ✓ Condition Respectée (sera détaillé dans la partie Tests unitaires)

Design du projet

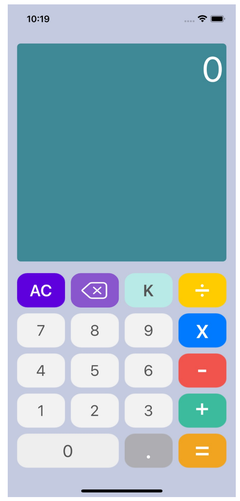
Le Design de l'application a été finalisé en reprenant l'existant mais pour lui donner un côté plus moderne et fonctionnel des ajustements ont été effectués.

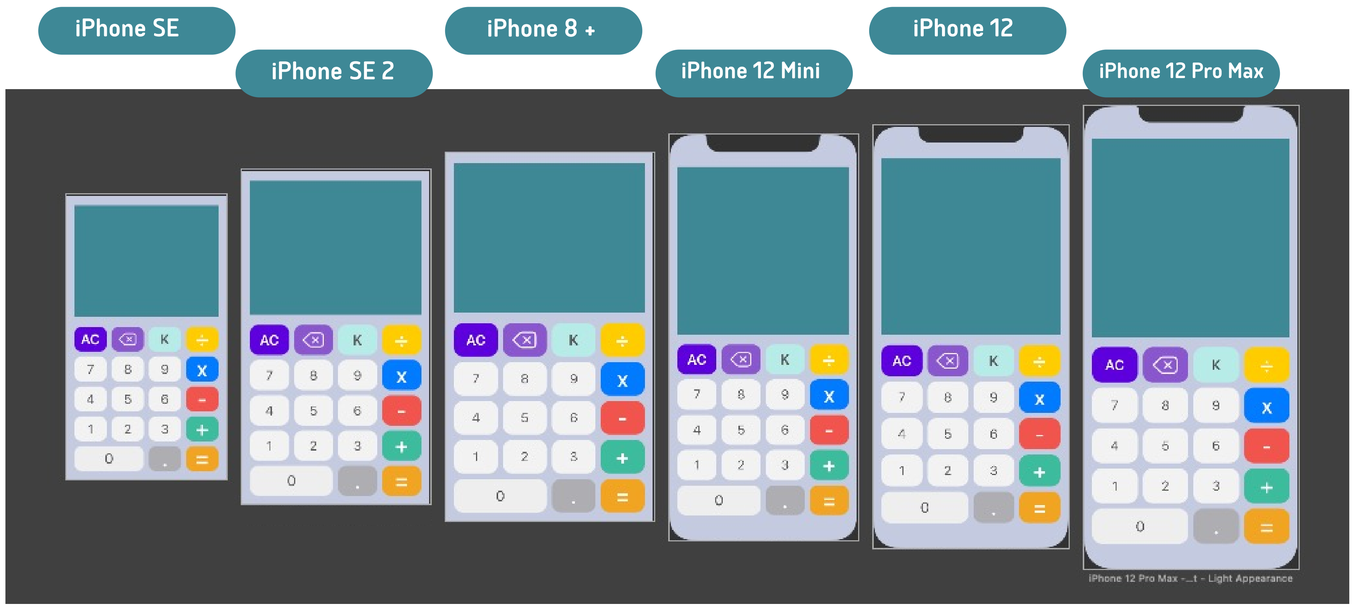
La disposition des chiffres a été revue pour correspondre aux standards d'une calculatrice de même pour les opérateurs.

Enfin l'ergonomie des boutons a été légèrement modifiées pour apporter un côté plus convivial à l'utilisateur.

Design Avant

Design Après





Glossaire

– : Private
: Internal
→ Interactions envoi et retour d'informations

❖ Architecture de l'application

L'application s'appuie sur le design pattern MVC. Ce qui signifie que la vue et le modèle ne communiquent jamais directement ensemble. Ils n'ont aucune connaissance l'un de l'autre.

La vue c'est notre interface avec laquelle l'utilisateur va interagir.

Elle va contenir l'UITextVIew, les UIButton, tout ce qui concerne les éléments graphiques.

Elle communique avec le contrôleur.

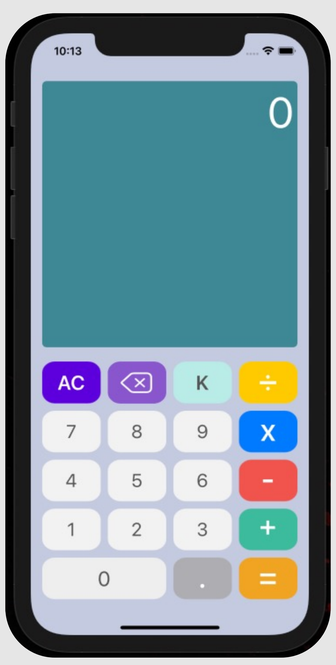
Le contrôleur lui apporte les modifications à la vue, ici l'affichage de notre opération et son résultat.

Le contrôleur va gérer l'intéraction entre la vue et le modèle. C'est lui qui permet la communication entre la Vue et le Modèle.
Lorsque les données changent, il va recevoir un événement du Modèle et envoyer un événement à la Vue.

Le modèle est le cerveau de l'application, il s'occupe du traitement des données.
Il s'occupe donc de la partie logique.
Il reçoit les informations, fait les calculs puis passe cela au contrôleur.

Il est indépendant de l'interface.

La Vue



La vue détecte qu'un bouton a été appuyé, elle passe l'information au contrôleur

1

6

La vue affiche le résultat à l'écran dans l'UITextView ou une Alerte reçue depuis le protocole ici ce sera 9

Le Contrôleur

ViewController: UIViewController

- @IBOutlet textView: UITextView!
@IBOutlet calculaterButtons: [UIButton]!

- calculator = Calculator()

- @IBAction tappedAllClearButton
- @IBAction tappedCorrectionButton
- @IBAction tappedKeepButton
- @IBAction tappedNumberButton
- @IBAction tappedDecimalButton
- @IBAction tappedOperatorsButton
- @IBAction tappedEqualButton

viewDidLoad

Extension ViewController: CalculatorDelegate

func updateDisplay
func showAlertPopUp

Le contrôleur interroge le modèle via l'IBAction associée au bouton en transmettant les informations nécessaires afin de mettre à jour l'opération en cours

2

5

Le contrôleur reçoit le résultat via le Delegate et il envoie à la vue les résultats ou l'alerte.

Le Modèle

Protocol CalculatorDelegate

func updateDisplay
func showAlertPopUp

Calculator

ErrorCase

delegate: CalculatorDelegate

calculInProgress
- operation
- firstValue
- canAddOperator
- expressionHaveEnoughElement
- lastElementDecimal
- operationHaveResult
- divideByZero

errorMessage
appendSelectedNumber
isDecimal
appendOperand
correction
reset
keepResult
makeOperation
- calculate
- formatDecimal

Le modèle reçoit les informations et exécute les opérations et calculs correspondants.

3

4

Le modèle passe les résultats ou l'alerte obtenue au contrôleur si le calcul a une erreur

Détail d'une exécution de calcul

Exemple : pour l'opération Suivante : 3 + 3 x 4 ÷ 2

Les méthodes appelées sont les suivantes :

appendSelectedNumber
appendOperand

L'affichage s'actualisera via la méthode CalculatorDelegate

Une fois que l'utilisateur presse le bouton Egal, l'information passe au contrôleur via l'IBOutlet

Puis elle passe du contrôleur vers le modèle qui va effectuer une autre série d'actions :
Ici l'IBAction demande l'exécution de la fonction : makeOperation

Celle-ci va au préalable vérifier si l'opération est Valide (le nombre d'éléments minimum, que l'opération ne finit pas par un opérateur, qu'elle est possible (on a pas de division par 0), qu'elle n'a pas déjà été calculée (Egal déjà cliqué et donc on a déjà un résultat)

Ensuite on passe au coeur de l'opération
Le calculInProgress est réécrit dans operation
La méthode makeOperation via vérifier l'existence ou non d'un opérateur de priorité (X ou ÷)

L'opération s'exécute jusqu'à ce qu'elle ne comporte plus qu'un élément qui sera le résultat de notre operation
Le résultat est ensuite formaté pour afficher un nombre entier si c'est un entier

Le modèle met à jour la propriété calculInProgress

❖ Code du projet

Utilisation d'un protocole

Un protocole définit un schéma de méthodes et de propriétés auxquels la classe va se conformer. Ce sont des exigences qui devront être respectées.
Dans notre cas on veut que ce protocole fasse 2 choses qui seront la mise à jour de l'affichage via **updateDisplay** et les alertes via **showAlertPopUp**

```swift
// Update View and Alert
protocol CalculatorDelegate: class {
    func updateDisplay(_ calculInProgress: String)
    func showAlertPopUp(title: String, message: String)
}
```

```swift
// MARK: - Properties

weak var delegate: CalculatorDelegate?

// Display operation on screen
var calculInProgress: String = "0" {
    didSet {
        delegate?.updateDisplay(calculInProgress)
    }
}
```

Dans notre classe **Calculator** on va donc créeer une instance de **CalculatorDelegate** et lui attribuer les propriétés et méthodes nécessaires en l'occurrence :

- **calculInProgress** pour le calcul actuel et qui sera affiché à l'écran
- **errorMessage** pour les alertes qui seront affichées à l'écran

```swift
// Display error message for the actual alert
func errorMessage(alert: ErrorCase) {
    delegate?.showAlertPopUp(title: alert.title, message: alert.message)
}
```

Comme on le sait en MVC le **Modèle** et la **Vue** ne communiquent jamais directement, on va donc avoir besoin que le **Controller** réponde au protocole afin de traiter la demande.
On va lier notre **Controller** à ce protocole et pour se faire on utilise une **Extension**.
Les **extensions** permettent de rajouter des fonctionnalités à un type ici notre classe **ViewController**.
Dans cette **extension**, on implante les paramètres nécessaires pour que tout fonctionne.

```swift
extension ViewController: CalculatorDelegate {
    func updateDisplay(_ calculInProgress: String) {
        textView.text = calculInProgress
        textView.layer.cornerRadius = 6
    }

    func showAlertPopUp(title: String, message: String) {
        let alert = UIAlertController(title: title, message: message, preferredStyle: .alert)
        alert.addAction(UIAlertAction(title: "OK", style: UIAlertAction.Style.default, handler: nil))
        self.present(alert, animated: true, completion: nil)
    }
}
```

Dans notre **extension**, la fonction **updateDisplay** créé le lien entre le **textView** et le **calculInProgress**.
La fonction **showAlertPopUp** quant à elle nous permet d'implémenter **l'UIAlertController** qui va afficher les messages et lier les paramètres de notre fonction (title & message)

Le **Model** communique donc au **Controller** les informations pour l'affichage ou les alertes via le delegate qui va les transmettre à la **Vue**.
Le delegate nous permet de déléguer la responsabilité de l'affichage du **Model** au **Controller**

Calculator

Notre Calculator est donc le cerveau de l'application, c'est lui qui s'occupe de la partie opérationnelle de notre calculatrice, il va se découper en plusieurs grands axes de méthodes et propriétés.

■ Ajout des éléments

On a séparé les fonctions d'ajouts d'éléments en 3, l'une pour l'ajout des nombres, la seconde pour le décimal et la dernière pour les opérateurs.

```swift
// Adding number to the operation
func appendSelectedNumber(_ numberText: String) {
    guard !operationHaveResult else { return errorMessage(alert: .operationHaveResult) }
    if calculInProgress == "0" {
        calculInProgress.removeLast()
        calculInProgress.append(numberText)
    } else {
        calculInProgress.append(numberText)
    }
}

// Adding decimal to the operation
func isDecimal() {
    guard operation.first?.contains("") == false else { return errorMessage(alert: .syntax) }
    guard operation.last?.contains(".") == false else { return errorMessage(alert: .decimalExist) }
    guard canAddOperator else { return errorMessage(alert: .syntax) }

    calculInProgress.append(".")
}
```

```swift
// Adding operand to the operation
func appendOperand(_ operandChoice: String) {
    guard !operationHaveResult else { return errorMessage(alert: .operationHaveResult) }
    guard !firstValue else { return errorMessage(alert: .operationImpossible) }
    guard lastElementDecimal else { return errorMessage(alert: .syntax) }
    if canAddOperator {
        switch operandChoice {
        case "+":
            calculInProgress.append(" + ")
        case "-":
            calculInProgress.append(" - ")
        case "x":
            calculInProgress.append(" x ")
        case "÷":
            calculInProgress.append(" ÷ ")
        default: errorMessage(alert: .operationImpossible)
        }
    } else {
        errorMessage(alert: .wrongOperator)
    }
}
```

Chaque fonction nous permet de vérifier un certain nombre de critères qui vont permettre où interdire l'écriture d'un calcul qui serait incorrect et donc susceptible de produire un crash de l'application.

■ Les propriétés

Comme vu précédemment, on a des critères de vérifications. Ce sont des propriétés qui ont été créée pour vérifier certaines actions de l'utilisateur lorsqu'elle se produisent. Ces propriétés sont appelées dans les fonctions à l'aide du Guard Statement, il permet de vérifier que cette propriété existe, se produit et permet l'exécution de l'action ou le cas échéant de retourner une valeur.

```swift
// Check is first element is a number
private var firstValue: Bool {
    if calculInProgress == "0" || calculInProgress == "" {
        return operation.first != "+" && operation.first != "-" &&
            operation.first != "÷" && operation.first != "x"
    }
    return false
}

// Check if last element in operation is an operator
private var canAddOperator: Bool {
    return operation.last != "+" && operation.last != "-" && operation.last != "÷" &&
        operation.last != "x"
}

// Check if we have 3 necessary elements for calcul
private var expressionHaveEnoughElement: Bool {
    guard operationHaveResult == false else { return true }
    return operation.count >= 3
}

// Check last element is not a Decimal
private var lastElementDecimal: Bool {
    return operation.last?.last != "."
}

// Operation have a result
private var operationHaveResult: Bool {
    return calculInProgress.firstIndex(of: "=") != nil
}

// Prevent divide by 0
private var divideByZero: Bool {
    return calculInProgress.contains("÷ 0")
}
```

Dans le cas de notre calculatrice, celui-ci permet de retourner les alertes liées à chaque type d'erreur.

■ Les Fonctions Opérantes

Celles-ci sont découpées en 2 fonctions, une qui gère l'ensemble de l'opération et la seconde qui va s'occuper de chaque phase du calcul

```
// Reduce all the operation and make full calculation
func makeOperation() {
    guard expressionHaveEnoughElement else { return errorMessage(alert: .operationImpossible) }
    guard canAddOperator, !operationHaveResult else { return errorMessage(alert: .operationHaveResult) }
    guard !divideByZero else { return errorMessage(alert: .divideByZero) }
    guard lastElementDecimal else { return errorMessage(alert: .syntax) }

    var operationsToReduce = operation
    let priorityOperators = ["x", "÷"]
    let classicOperators = ["+", "−"]
    var currentIndex: Int?
    var result = ""

    while operationsToReduce.count > 1 {

        // Checking priority operators
        let indexPriorityOperator = operationsToReduce.firstIndex(where: {priorityOperators.contains($0)})
        if let operatorActivePrior = indexPriorityOperator {
            currentIndex = operatorActivePrior
        } else {
            // Classic operators
            let indexClassicOperator = operationsToReduce.firstIndex(where:
                {classicOperators.contains($0)})
            if let operatorActiveClassic = indexClassicOperator {
                currentIndex = operatorActiveClassic
            }
        }

        // Execute method to make calculation by verify index
        if let index = currentIndex {
            let operand = operationsToReduce[index]
            guard let onLeft = Double(operationsToReduce[index - 1]) else { return }
            guard let onRight = Double(operationsToReduce[index + 1]) else { return }
            result = formatDecimal(number: calculate(leftNumber: onLeft, operand: operand, rightNumber:
                onRight))
            operationsToReduce[index] = result
            operationsToReduce.remove(at: index + 1)
            operationsToReduce.remove(at: index - 1)
            calculInProgress.removeAll()
        }
        calculInProgress.append(" = \(operationsToReduce[0])")
    }
}
```

On utilise de nouveau les guard pour vérifier la cohérence du calcul

Notre opération affichée à l'écran est réécrite en supprimant les espaces et en la transformant pour qu'elle devienne un tableau.

```
private var operation: [String] {
    return calculInProgress.split(separator: " ").map { "\($0)" }
}
```

De cette manière notre processus de réduction sera applicable, le but est d'arriver à une opération qui va contenir un seul élément qui sera le résultat de notre opération.

Comme nous le savons en Mathématiques la multiplication et la division sont prioritaires sur l'addition et la soustraction.

On vérifie donc dans un premier temps s'il existe des opérateurs de priorité, si c'est le cas l'opération se poursuit sur ceux-ci

Dans le cas contraire on passe sur les opérateurs courants que sont le + et le -

Notre calcul traitera toujours en priorité les opérateurs de priorité avant les opérateurs courants.

Dès qu'un opérateur est repéré, on va pouvoir exécuter le calcul mathématique pour cela on se réfère à l'index de l'opérateur et on interroge les 2 éléments à proximité, à gauche: index - 1 et à droite: index + 1

Le calcul s'exécute et les éléments utilisés sont supprimés au fur et à mesure pour ne laisser que le résultat.

Enfin on ajoute le résultat au calculInProgress pour que le controller le renvoie à la vue

```
// Method to calculate when operand
private func calculate(leftNumber: Double, operand: String, rightNumber: Double) -> Double {
    var result: Double = 0.0
    switch operand {
    case "+": result = leftNumber + rightNumber
    case "−": result = leftNumber − rightNumber
    case "x": result = leftNumber * rightNumber
    case "÷": result = leftNumber / rightNumber
    default: break
    }
    return result
}
```

```
// Format method to remove decimal
private func formatDecimal(number: Double) -> String {
    let format = NumberFormatter()
    format.minimumFractionDigits = 0
    format.maximumFractionDigits = 5
    guard let value = format.string(from: NSNumber(value: number)) else { return ""}
    return value
}
}
```

On utilise la méthode formatDecimal pour formater notre Double en String et par la même occasion définir le nombre de décimales maximales sur un nombre non entier. Si celui-ci est un entier le nombre de décimales est à Zéro.

■ Les Fonctionnalités améliorées

```swift
// Correction elements
func correction() {
    guard !operationHaveResult else { return errorMessage(alert: .operationHaveResult)}
    if calculInProgress.last == " " {
        calculInProgress = String(calculInProgress.dropLast(3))
    } else {
        calculInProgress = String(calculInProgress.dropLast())
    }
}

// Remove all elements
func reset() {
    calculInProgress.removeAll()
    delegate?.updateDisplay("0")
}

// To keep last result for an other operation
func keepResult() {
    guard operationHaveResult, let lastResult = operation.last else { return errorMessage(alert: .keeping) }
    calculInProgress = lastResult
}
```

Le but était d'apporter à la calculatrice des fonctionnalités utiles dans le cadre de son utilisation.
La première s'occupe de la correction d'un calcul tant que celui-ci n'a pas de résultat.
La seconde sert à redémarrer un calcul à Zéro.
Et la dernière nous permet de garder le résultat d'une opération précédente pour la réutiliser.

■ Les Alertes

```swift
// Display error message when operations errors
enum ErrorCase {
    case operationImpossible, operationHaveResult, wrongOperator, divideByZero, decimalExist, keeping, syntax
    var title: String {
        switch self {
        case .operationImpossible: return "Error"
        case .operationHaveResult: return "Warning"
        case .wrongOperator: return "Error"
        case .divideByZero: return "Error"
        case .decimalExist: return "Warning"
        case .keeping: return "Error"
        case .syntax: return "Correction"
        }
    }
    var message: String {
        switch self {
        case .operationImpossible: return "This operation is impossible"
        case .operationHaveResult: return "This operation have a result. Press K to Keep last result or AC to clear"
        case .wrongOperator: return "An operand already exist !"
        case .divideByZero: return "You try to divide by 0. This operation is impossible"
        case .decimalExist: return "A decimal was already added to the operation"
        case .keeping: return "Nothing to keep"
        case .syntax: return "Syntax error, you need to correct the operation"
        }
    }
}
```

Elles sont là pour avertir l'utilisateur de son erreur mais aussi le guider le cas échant dans sa résolution.

ViewController

```swift
final class ViewController: UIViewController {

    // MARK: - @IBOUtlet

    @IBOutlet private weak var textView: UITextView!
    @IBOutlet var calculaterButtons: [UIButton]!

    // MARK: - Private Property

    private var calculator = Calculator()
```

```swift
    // MARK: - @IBAction

    @IBAction private func tappedAllClearButton(_ sender: UIButton) {
        calculator.reset()
    }

    @IBAction private func tappedCorrectionButton(_ sender: UIButton) {
        calculator.correction()
    }

    @IBAction private func tappedKeepButton(_ sender: UIButton) {
        calculator.keepResult()
    }

    @IBAction private func tappedNumberButton(_ sender: UIButton) {
        guard let numberText = sender.title(for: .normal) else { return }
        calculator.appendSelectedNumber(numberText)
    }

    @IBAction private func tappedDecimalButton(_ sender: UIButton) {
        calculator.isDecimal()
    }

    @IBAction private func tappedOperatorsButton(_ sender: UIButton) {
        guard let operandChoice = sender.title(for: .normal) else { return }
        calculator.appendOperand(operandChoice)
    }

    @IBAction private func tappedEqualButton(_ sender: UIButton) {
        calculator.makeOperation()
    }
}
```

Notre ViewController fait le lien entre la Vue et le Modèle

Les IBOutlet sont la connexion entre un objet de la vue et le controller

Ils sont au nombre de deux ici :

L'un pour l'UITextView qui va permettre l'affichage du calcul
L'autre est une UIButton Collection pour gérer les boutons de la calculatrice

Les IBAction vont définir l'évènement associé à un élément
A chaque pression sur un bouton une action se déclenche qui va appeler une fonction du Model

Pour le tappedNumberButton et le tappedOperatorsButton on à un guard qui va vérifier que l'opérateur ou le nombre est en adéquation avec le titre du bouton.
Ainsi si le + est tapé on aura l'ajout du + dans l'opération comme voulu.

```swift
    // MARK: - View Life cycles

    override func viewDidLoad() {
        super.viewDidLoad()
        calculator.delegate = self

        textView.layer.cornerRadius = 6
        textView.text = calculator.calculInProgress
    }
```

Enfin notre viewDidLoad permet le chargement d'éléments spécifique avant le démarrage de l'application

Tests unitaires

Comme vu précédemment on a des propriétés créées pour détecter les erreurs mais effectuer des tests manuels serait long et potentiellement sujet à l'erreur.
De plus le jour où l'on effectue des modifications sur une application, les tests unitaires nous permettent de savoir rapidement si notre application ne va pas rencontrer des bugs.

On a donc recours aux tests unitaires qui vont automatiser cela.

On a commencé par tester les formules mathématiques et chaque élément des opérations les plus simples aux plus complexes

```swift
import XCTest
@testable import CountOnMe

final class CalculatorTests: XCTestCase {

    var calculator: Calculator!
    var calculatorDelegateMock = CalculatorDelegateMock()

    override func setUp() {
        super.setUp()
        calculator = Calculator()
        calculator.delegate = calculatorDelegateMock
    }
```

```swift
// Multiplication
func testGiven6Multiplying3_WhenMultiplication_ThenResultSouldBe18() {
    calculator.appendSelectedNumber("6")
    calculator.appendOperand("x")
    calculator.appendSelectedNumber("3")
    calculator.makeOperation()
    XCTAssertEqual(calculator.calculInProgress, " = 18")
}

// Division
func testGiven10Divide2_WhenDivision_ThenResultSouldBe5() {
    calculator.appendSelectedNumber("10")
    calculator.appendOperand("÷")
    calculator.appendSelectedNumber("2")
    calculator.makeOperation()
    XCTAssertEqual(calculator.calculInProgress, " = 5")
}

// Test Complexe Operation
func testComplexeOperation_WhenAllOperand_ThenResultSouldBe67Dot9() {
    calculator.calculInProgress = "40 + 6 x 9.3 ÷ 2"
    calculator.makeOperation()
    XCTAssertEqual(calculator.calculInProgress, " = 67.9")
}
```

Puis ensuite on a testé les alertes

On a donc créé un Mock pour le delegate

Il s'agit d'un objet simulé qui va reproduire le comportement de l'objet réel.
On va donc pouvoir tester le comportement des alertes

```swift
import XCTest
@testable import CountOnMe

class CalculatorDelegateMock: CalculatorDelegate {
    var testAlertTitle: String = ""
    var testAlertMessage: String = ""

    func updateDisplay(_ calculInProgress: String) {
    }

    func showAlertPopUp(title: String, message: String) {
        testAlertTitle = title
        testAlertMessage = message
    }
}
```

```swift
// MARK: - Test Alert

// Alert enough Elements Test
func testOperationIsImpossible_WhenOperationHaventEnoughElements_ThenResultSouldBeFalse() {
    calculator.appendSelectedNumber("2 + ")
    calculator.makeOperation()
    XCTAssertEqual(calculator.calculInProgress, "2 + ")
    XCTAssertEqual(calculatorDelegateMock.testAlertTitle, Calculator.ErrorCase.operationImpossible.title)
    XCTAssertEqual(calculatorDelegateMock.testAlertMessage, Calculator.ErrorCase.operationImpossible.message)
}

// Alert Division by 0
func testGivenAlertMessage_WhenLastElementsAreDivideandZero_ThenResultSouldBeDisplayAlertMessage() {
    calculator.calculInProgress = "5 ÷ 0"
    calculator.makeOperation()
    XCTAssertEqual(calculatorDelegateMock.testAlertMessage, Calculator.ErrorCase.divideByZero.message)
}
```

On arrive donc à un Code Coverage de 100% sur notre Calculator qui nous assure que notre code est correctement testé et donc fonctionnel.