



# PE - Apprentissage du langage Swift

Semestre A22



utt  
UNIVERSITÉ DE TECHNOLOGIE  
TROYES

# Sommaire

<b>Sommaire</b>	<b>2</b>
<b>1 - Introduction</b>	<b>2</b>
1.1 Contexte	2
1.2 Présentation du sujet	2
1.3 Problématique	2
<b>2 - Environnement de développement Apple</b>	<b>4</b>
2.1 Présentation du langage Swift	4
2.1.1 Histoire du langage	4
2.1.2 Le playground	5
2.1.3 La bibliothèque UIKit	7
2.1.4 La bibliothèque SwiftUI	9
2.2 Le modèle MVC - Model View Controller	10
2.3 Le modèle MVVM - Model View ViewModel	12
<b>3 - Contributions</b>	<b>12</b>
3.1 Design Pattern 1 : Le Model View Controller	12
3.1.1 L'arborescence de fichiers	13
3.1.2 Création du ou des modèles	13
3.1.3 La création du storyboard et la liaison avec le Controller	13
3.1.4 Le cerveau de l'application : le Controller	13
3.1.5 Conclusion sur l'implémentation du modèle	15
3.2 Design Pattern 2 : Le Model View ViewModel	16
3.2.1 Création de l'application	16
3.2.2 Implémentation du ViewModel	18
3.2.3 Fonctionnement de l'application	19
3.2.4 Conclusion sur l'implémentation du modèle	20
3.3 Synthèse et point de vue	20
<b>4 - Conclusion</b>	<b>21</b>
4.1 Les missions et contraintes	21
4.2 Les contributions	22
4.3 Ouverture du sujet	22
<b>Bibliographie</b>	<b>23</b>
2 - Environnement de développement Apple	23
3 - Contributions	23
3.1 Design Pattern 1 : Model View Controller MVC	23
3.2 Design Pattern 2 : Model View ViewModel	23
<b>Annexes</b>	<b>23</b>

# 1 – Introduction

## 1.1 Contexte

J'ai choisi de réaliser un PE de développement d'application mobile car c'est un domaine de programmation que je ne connaissais pas pour lequel j'ai de l'intérêt. Je me suis dirigé vers la réalisation d'applications iOS pour découvrir le langage Swift, souvent présenté comme un langage moderne et innovant. Enfin, pour approfondir le projet j'ai choisi, sur conseil de mon tuteur, d'étudier des design patterns architecturaux applicables lors du développement d'applications iOS afin d'en comprendre leur utilité.

## 1.2 Présentation du sujet

Le but de ce projet étudiant est de se former en autonomie encadrée (apprentissage seul avec un suivi régulier par un enseignant) au développement mobile.

Le projet est centré sur l'utilisation des design patterns architecturaux, patrons de conceptions en français. En voici une définition : *"An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context"*. Le but est donc d'établir une méthode générale de création d'application mobile, qui soit fiable et maintenable. Les design patterns architecturaux sont une boîte à outils qui pourraient le permettre.

## 1.3 Problématique

Le but du projet étudiant est donc d'étudier le développement d'applications mobiles iOS grâce aux solutions proposées par Apple : Swift, ses bibliothèques et ses design patterns. On peut alors établir la problématique suivante :

**Comment réaliser des applications mobiles sûres, viables et bien codées ?**

## 2 – Environnement de développement Apple

### 2.1 Présentation du langage Swift

#### 2.1.1 Histoire du langage

Le langage Swift est un langage lancé en 2014 par Apple ayant pour but de créer des applications mobiles simples et sécurisées. Le langage se veut rapide et moderne. Il est open-source (depuis 2015) et mis à jour régulièrement : la dernière version est Swift 5.7, paru en septembre 2022 .

À sa sortie, Swift est bien reçu par la communauté de développeurs, habituée à d'autres langages, qui participe à l'amélioration du langage en faisant de nombreux retours dès sa sortie. Swift devient alors un langage très populaire voire incontournable du développement mobile.

Swift est influencé par plusieurs autres langages plus anciens : Objective-C, C#, Haskell... Parmi ses contributeurs principaux on peut citer Chris Lattner qui a initié le développement du langage. Cependant, comme le langage est open-source depuis 2015, il compte de nombreux autres contributeurs (Google et IBM en font même partie). Le langage Swift est conçu pour être accessible à des développeurs débutants notamment grâce à la mise en place du playground (nous reviendrons dessus plus tard).

Sur des langages comme le C ou C#, la mémoire doit être gérée par le développeur. Cela peut poser de nombreux problèmes comme des fuites de mémoire, de l'instabilité ou un manque de sécurité de l'application ou du programme. C'est pour cette raison que Swift dispose d'une gestion automatique de la mémoire qui garantit une plus grande rapidité d'exécution et une meilleure sécurité.

Le langage possède néanmoins un inconvénient remarquable : sa fermeture. Cela peut paraître ironique pour un langage de programmation open-source mais Apple garde un contrôle indirect sur le langage. En effet, il permet de développer des applications sur les appareils d'Apple, donc si on souhaite coder une application il faut utiliser la version de Swift officielle donnée par Apple. C'est notamment cette fermeture qui a fait que Chris Lattner a quitté Apple après le lancement du langage.

Swift est complété par de nombreuses bibliothèques, souvent indispensables à la création d'applications. Les plus connues (et officielles) sont UIKit, Foundation et la plus récente : SwiftUI.

## 2.1.2 Le playground

Le playground est le terrain de jeu des développeurs Swift. Il permet aux débutants comme aux plus expérimentés de tester des fonctions, des nouveautés en quelques clics. C'est généralement le point de départ pour développer en Swift et j'ai donc commencé ce projet étudiant en explorant les différentes possibilités qu'offrent ce langage.

En m'appuyant sur un TD de l'UE IF26 j'ai rédigé un document retraçant les différentes fonctionnalités du langage Swift (uniquement le langage, pas les bibliothèques) (voir annexes 1 et 2). Ce document a pu m'être utile à de multiples reprises dans la suite de mon projet étudiant lorsque que je souhaitais vérifier la syntaxe de mon code (boucles, affichages). Voici une énumération non exhaustive d'éléments de syntaxe du langage.

Pour déclarer un objet, on choisit si on lui applique la contrainte d'être constant ou pas avec le préfixe "let". Par exemple pour le code ci-dessous, l'instruction **y=x** est impossible.

```
var x = 2
let y = 3
```

Figure 1 : exemple de déclaration de variable et de constante

Dans le cas ci-dessous, nous n'avons pas précisé le type de la variable, Swift en a déduit tout seul qu'il s'agissait d'entiers. Cependant pour limiter les erreurs il est préférable de déclarer systématiquement le type des variables.

```
var prix : Double = 12.56
let prenom: String = "julien"
```

Figure 2 : exemple de déclaration de variable

La déclaration et l'appel de fonction sur Swift est assez simple et ressemble à la syntaxe du C.

```
func echo1(chaine : String){
    print(chaine)
}
echo1(chaine : prenom)
```

```
func isAdult() -> Bool {
    if self.age < 18 {
        return false
    }
    else {
        return true
    }
}
```

Figures 3 et 4 : exemples de déclaration et appel de fonction en Swift

Pour déclarer des listes, des tuples des dictionnaires et les appeler on utilise les instructions ci dessous. Cette syntaxe est plus semblable à celle de Python.

```
let liste = ("IF26", "L007", "NF19")
print(liste.1)
let liste2 = (IF26: "Développement mobile", L007: "Dev Web", NF19 : "Cloud")
print(liste2.IF26)
```

Figure 5 : exemple de déclaration et appel de tuple et dictionnaire

Les classes sont indispensables en Swift, elles utilisent pleinement les fonctions d'un langage orienté objet : initialisation, déclaration de méthodes, variables locales. Voici ci-dessous un exemple complet. On peut voir que la classe traite deux cas distincts de déclaration : celui où l'utilisateur n'entre pas de valeur par défaut (dans la méthode **init()**) et celui où il déclare les valeurs (utilisation de la méthode **init (avec arguments)**). Dans le cas de l'exemple, c'est cette dernière qui est utilisée lors de la création de la variable **p1**.

```
class Personne {
    var nom: String
    var prenom: String
    var age: Int

    var description : String {
        return "Je m'appelle \(prenom) \(nom) et j'ai \(age) ans"
    }

    init() {
        self.nom = "?"
        self.prenom = "?"
        self.age = 0
    }

    init(nom : String, prenom : String, age : Int){
        self.nom = nom
        self.prenom = prenom
        self.age = age
    }

    func isAdult() -> Bool {
        if self.age < 18 {
            return false
        }
        else {
            return true
        }
    }
}

var p1 = Personne(nom : "Calonne", prenom : "Julien", age : 20)
p1.isAdult()
p1.description
```

Figure 6 : exemple de déclaration et d'utilisation de classe en Swift

On peut aussi créer des classes héritées d'une classe. Elles permettent de réutiliser les différentes méthodes de la classe "maître". Il y a deux instructions importantes à connaître lorsqu'on manipule des classes héritées en Swift : le préfixe

**super** permet d'accéder à une méthode de la classé "maître" et on utilise **override** devant le nom d'une méthode pour modifier une méthode existante.

Pour finir, Swift nous permet aussi d'utiliser des Closure (fermeture en français). Une closure est une suite d'instructions pouvant avoir des paramètres d'entrée et de sortie contenu dans une variable. Les Closure permettent alors d'utiliser une fonction sans l'avoir déclarée avant et peuvent être utilisées pour attendre une action de l'utilisateur (sans la connaître au préalable) ou d'attendre la fin d'un chargement.

```
let bonjourClosure = { () -> String in
    return "Bienvenue dans le premier Closure"
}
print(bonjourClosure())

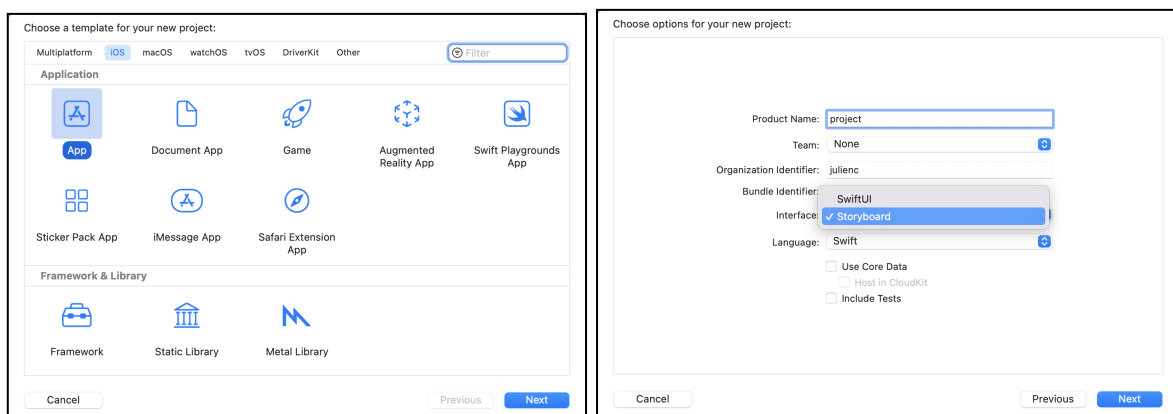
let addClosure = { (n1 : Int, n2: Int, n3: Int) -> Int in
    return n1+n2+n3
}
addClosure(1,2,3)
```

Figure 7: exemple de déclaration et d'appel de Closure

### 2.1.3 La bibliothèque UIKit

UIKit est une bibliothèque qui permet à l'utilisateur de créer une interface d'application en y "glissant" des composants. La bibliothèque fonctionne pour le développement multi-platform (iOS, iPadOS, tvOS).

Pour utiliser la bibliothèque et l'implémenter simplement, il suffit de créer notre application en sélectionnant l'interface storyboard, comme ci-dessous :



Figures 8 et 9 : création d'un projet utilisant UIKit

XCode crée alors le projet pour une vue unique dans un répertoire contenant tout le code et les ressources associées à l'application.

- Les deux premiers fichiers (*AppDelegate* et *SceneDelegate*) sont des fichiers de supports dont la modification n'est pas nécessaire pour créer une application. Je ne les ai pas utilisés pour mes contributions.

- Le fichier *ViewController* est le fichier dans lequel Apple nous indique qu'il faut créer les fonctions pour modifier notre vue.
- Le fichier *Assets* regroupe les ressources graphiques de notre application telles que les logo en différentes tailles ou toutes les images que nous pourrions utiliser qui sont visibles sur l'interface.
- Enfin, les fichiers *Main* et *LaunchScreen* sont des fichiers constituant les vues de notre application. Comme leurs noms l'indiquent, *LaunchScreen* correspond à l'écran de chargement de l'application et *Main* contient les différentes vues ainsi que leurs composants.

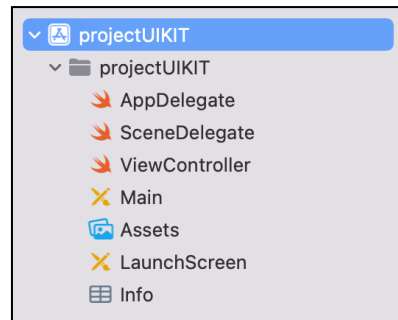


Figure 10 : Fichiers minimums nécessaires au développement d'une application utilisant UIKit

Le développement utilisant UIKit est centré sur le storyboard (ensemble des vues). C'est sur celui-ci que l'on glisse les composants (appelés Outlets par Apple) que l'on souhaite ajouter à notre vue. UIKit dispose d'une multitude de composants prédéfinis répondant à toutes les demandes des développeurs. Nous verrons dans la partie **3 - Contributions** comment paramétrer et modifier ces composants (et surtout dans quel fichier le faire) dans le cas où l'on utilise un design pattern.

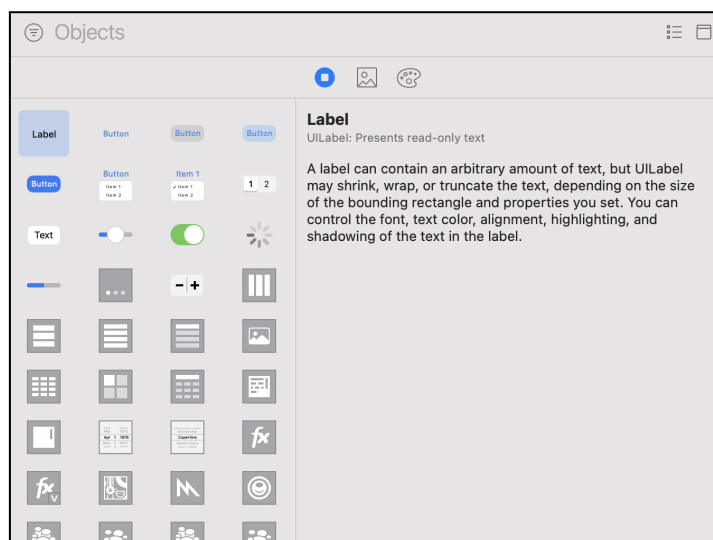


Figure 11 : capture d'écran de l'interface de recherche de composants.

La hiérarchisation des vues et des composants reste très précise car chaque élément est imbriqué dans un autre. Une arborescence comme celle ci-dessous



permet donc au développeur d'avoir une vision globale sur l'héritage car il peut appliquer une classe différente sur chaque vue ou sur une vue "maître" et ses sous vues.

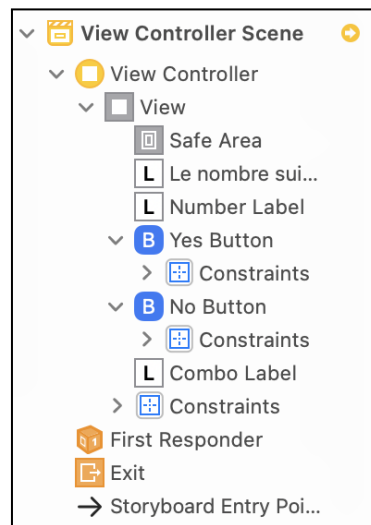


Figure 12 : capture d'écran de l'héritage/l'arborescence des vues et composants

Pour conclure, UIKit est une bibliothèque d'Apple complète et fonctionnelle limitant le code à "écrire". Elle apporte plus qu'une simple liste de fonctions précompilées à importer, elle permet au développeur de disposer de tout un écosystème de développement.

Cependant elle pourrait être amenée à être moins utilisée dans les années à venir à cause de la sortie en 2019 de la bibliothèque SwiftUI.

## 2.1.4 La bibliothèque SwiftUI

SwiftUI est une bibliothèque très récente par rapport à UIKit puisqu'elle est sortie en même temps qu'iOS 13 en 2019. Elle propose une nouvelle syntaxe et une nouvelle manière de réaliser des applications pour les développeurs.

SwiftUI est présentée par Apple comme étant plus simple à apprendre et à prendre en main. Cela est possible notamment grâce à l'apparition de la *Live Preview* qui permet de visualiser l'application que l'on code sans avoir à recharger l'émulateur d'iPhone sur lequel se situe l'application (ce qui était le cas avec UIKit).

Pour développer en utilisant SwiftUI il faut sélectionner **SwiftUI** au lieu de **storyboard** lors de la création du projet (vu dans la partie [2.1.3 La bibliothèque UIKit](#)). XCode crée alors une ContentView qui est la vue principale de l'application. Cette vue correspond à la zone entourée en bleue sur la figure 13.

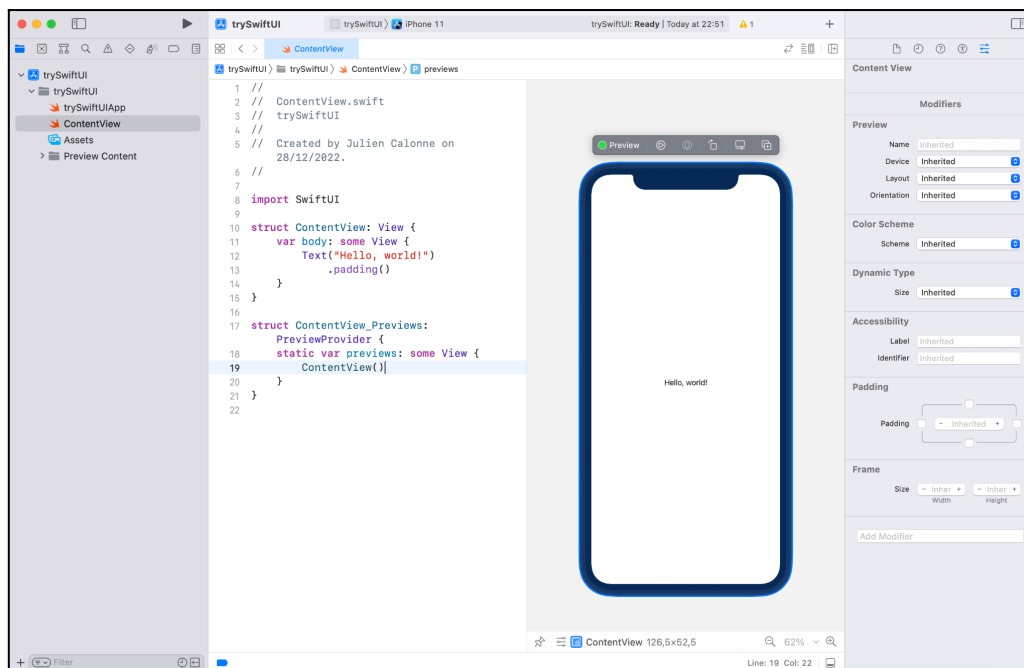


Figure 13 : Code à gauche, Live Preview à droite

Pour comprendre la différence entre les deux bibliothèques il faut comprendre la différence entre une application **Event Driven** et une application **State Driven**. Une application Event Driven va actualiser ses vues en fonction de l'apparition d'un événement. Tandis qu'une application State Driven va actualiser ses vues à chaque fois qu'une donnée est modifiée. L'application Event Driven nécessitera plus de code car il faut penser à tous les cas possibles. UIKit permet de faire des applications Event Driven et SwiftUI des applications State Driven.

Étant sorti fin 2019, SwiftUI reste une bibliothèque très récente et par conséquent sa communauté d'utilisateur est assez petite. Il y a donc moins de documentation à son propos et il est plus difficile de résoudre certaines erreurs rares. C'est pour cette raison que j'ai décidé d'étudier les Design Patterns en utilisant la bibliothèque UIKit.

## 2.2 Le modèle MVC – Model View Controller

Le modèle MVC est un modèle incontournable parmi les Design Patterns. Il est à la fois simple d'utilisation et robuste. La manière dont est conçu Swift et son IDE XCode nous incite à utiliser le modèle MVC. Par exemple, lorsqu'on crée un projet storyboard sur XCode, un fichier ViewController.swift est créé par défaut.

Pour créer une application en MVC on distingue donc 3 types de fichier : les **modèles**, les **vues** et les **controllers**.

Les **modèles** constituent les données de l'application. Ils sont formés par une classe ou une structure qui leur donne des attributs et des méthodes. Ils définissent des initialisateurs (valeurs par défaut). Il faut cependant faire attention au choix de type de modèle car les classes et les structures ne sont pas les mêmes. Les classes peuvent

hériter d'autres classes et sont des types de références alors que les structures sont des types de valeurs. Voici les définitions données par le site **appypie.com** :

*Types de valeur* : Lorsque vous copiez un type de valeur (c'est à dire lorsqu'il est attribué, initialisé ou transmis dans une fonction ), chaque instance conserve une copie unique des données. Si vous changez une instance, l'autre ne change pas aussi.

*Type de référence* : Lorsque vous copiez un type de référence, chaque instance partage les données. La référence elle-même est copiée, mais pas les données auxquelles elle fait référence. Lorsque vous changez l'un, l'autre change aussi.

Les **vues** sont la partie visible par l'utilisateur, c'est avec elles qu'il interagit. Elles permettent d'afficher tous les éléments transmis en continu par le Controller et dans un autre sens, de transmettre au Controller les actions réalisées par l'utilisateur.

Le **Controller** fait donc le lien entre les actions de l'utilisateur et les données contenues par l'application. C'est lui qui contient les fonctions qui permettent à l'application de fonctionner : c'est son cerveau. Il n'y a donc pas d'interaction entre les modèles et les vues en MVC (voir figure 14).

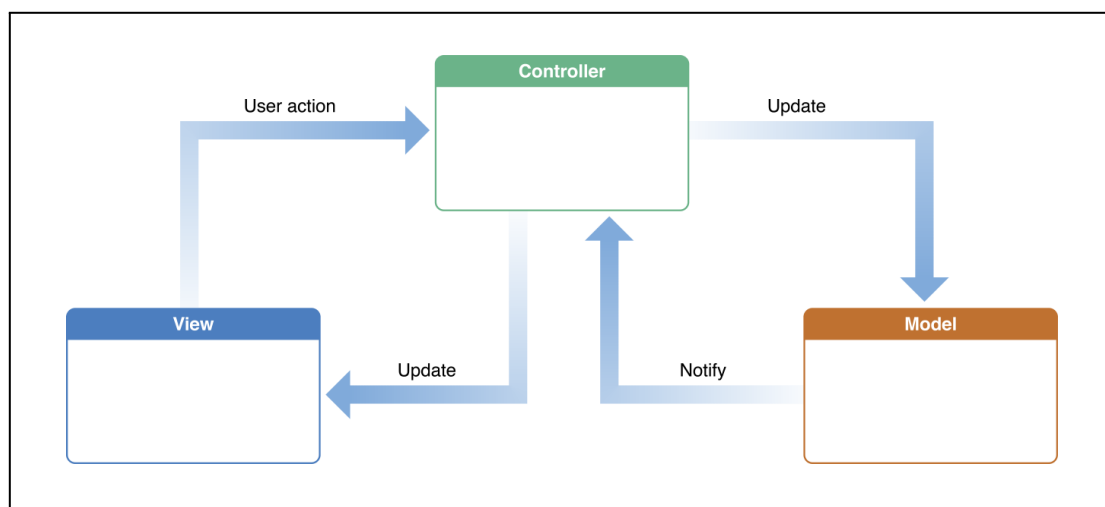


Figure 14 : Schéma récapitulatif du MVC donné par Apple

## 2.3 Le modèle MVVM – Model View ViewModel

Le MVVM est une variante du modèle MVC qui a pour but de séparer davantage les données ainsi que la logique de présentation des applications.

Le **modèle** en MVVM ne diffère pas du modèle MVC. Il sert à décrire les données et sa taille varie en fonction du projet. Le **ViewModel** récupère des données et les transmet en les mettant en forme pour ne conserver que ce qui est utile au Controller et ce qui est affiché à l'utilisateur. Il peut aussi contenir des informations sur des composants (si un bouton est pressé ou pas). Le ViewModel assure la connexion entre

un modèle et une vue. Le **Controller** s'occupe de gérer les interactions avec la **View**, de modifier les données et est constitué de différentes fonctions qui font fonctionner l'application. Il a une utilité similaire à celui du modèle MVC.

## 3 – Contributions

### 3.1 Design Pattern 1 : Le Model View Controller

Comme précisé dans la partie : [2 – Environnement de développement Apple](#), on peut choisir de créer son application avec deux bibliothèques différentes : UIKit ou SwiftUI. Ayant suivi un cours sur l'implémentation du modèle MVC en utilisant la bibliothèque UIKit, j'ai choisi de continuer à l'utiliser. Cependant, chaque application est réalisable selon nos préférences, il n'y a pas de "norme".

L'application que j'ai choisi de créer pour le modèle MVC est un jeu : l'application propose un nombre aléatoire sur l'écran à un joueur et celui-ci doit dire s'il est premier ou non. Le jeu est basique et pour qu'il puisse permettre de comprendre l'architecture et le fonctionnement du modèle MVC sans difficulté.

Dans la suite des sous-parties je présente la manière que j'ai utilisée pour réaliser cette application le plus simplement possible, selon moi, en utilisant le modèle. Il existe sans doute plusieurs autres méthodes pour le réaliser autrement en MVC.

#### 3.1.1 L'arborescence de fichiers

Tout d'abord, il est important d'organiser l'arborescence des fichiers de notre application pour être cohérent avec le modèle MVC dès la création du projet sur XCode. On peut voir sur les captures d'écran suivantes la différence d'architecture d'application avant même d'avoir écrit une ligne de code, simplement la création du projet. La première étape de réalisation d'un projet MVC est donc de créer les dossiers Model, View et Controller.

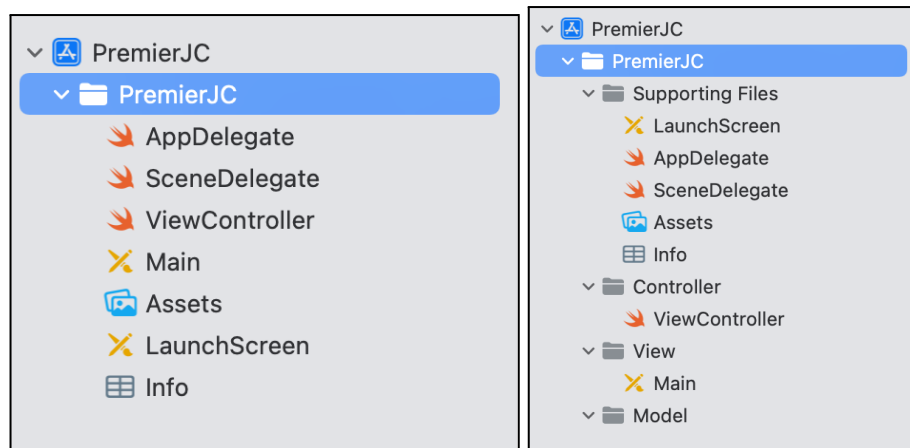


Figure 15 : Comparaison d'arborescence (sans MVC à gauche, avec MVC à droite)

### 3.1.2 Création du ou des modèles

Dans le cas de notre application de nombre premier nous n'aurons besoin que d'un modèle : **Number**. C'est une structure possédant deux attributs : sa valeur (Int) et s'il est premier (Bool). Ces valeurs sont initialisées respectivement à 0 et false et seront modifiées dans le controller par la suite.

```

8  import Foundation
9
10 struct Number {
11     var value = 0
12     var isPrime = false
13 }

```

Figure 16 : Structure Number dans Model

### 3.1.3 La création du storyboard et la liaison avec le Controller

Le storyboard constitue notre vue principale. Il est composé de plusieurs éléments de la bibliothèque UIKit (UILabel, UIButton). Tous ces éléments sont ajoutés et modifiés (visuellement) avec des "glisser-déposer" depuis le menu "+" du storyboard dans XCode. Il n'y a donc pas de code apparent associé à cette vue.

Ici, l'interface est simpliste : l'utilisateur comprend vite le principe du jeu. Il doit cliquer sur Yes ou No en fonction de s'il pense que le nombre affiché est premier ou non. En fonction de la validité de sa réponse, la valeur de combo est incrémentée ou remise à 0.



Figure 17 : Interface visible par l'utilisateur

### 3.1.4 Le cerveau de l'application : le Controller

Nous avons ensuite créé un modèle et une vue mais il faut faire fonctionner les deux ensemble pour que l'application soit utilisable, nous allons donc créer le Controller.

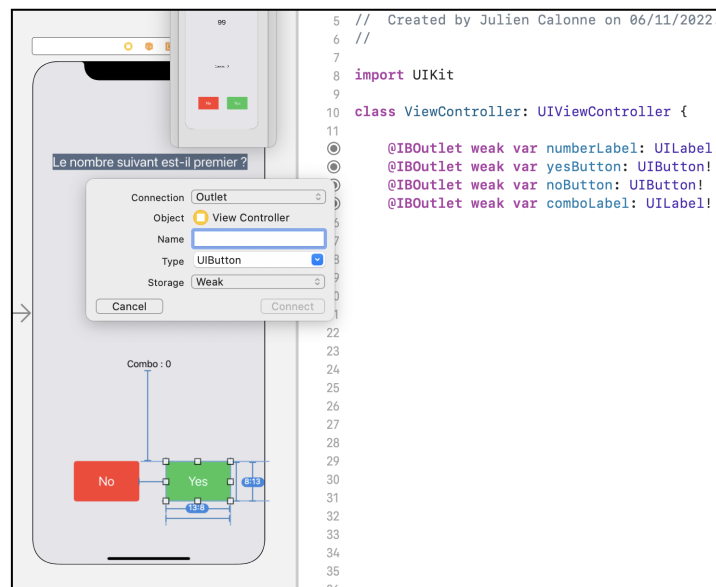


Figure 18 : capture d'écran des créations d'@IBOutlets

La première question que le développeur se pose est : comment relier ces éléments d'UIKit à des variables modifiables dans le Controller ? Pour cela, encore une fois Xcode simplifie la tâche. Il suffit d'appuyer sur "control" et de glisser déposer le bouton de la vue dans le Controller. Une fenêtre apparaît, elle reconnaît le type de l'élément et propose d'ajouter un nom (voir figure 18).

En suivant cette approche pour chaque élément on obtient des variables unwrapped (obligatoirement déclarées, on les reconnaît au point d'exclamation) dont on peut modifier les propriétés.

Ensuite, j'ai pu créer différentes fonctions qui servent à faire fonctionner le programme. Voici une liste des fonctions présentes dans le controller :

```
func newNumber()  
func randomInt(n: Int) -> Int  
func isPrime(number: Int) -> Bool  
private func answerQuestion(answer: Bool)|
```

Figure 19 : Ensemble des fonctions créées dans le Controller

La plupart des noms des fonctions sont assez explicites quant à leurs utilités (modifier les valeurs de la vue, les calculer...) mais nous allons nous attarder sur la fonction `answerQuestion()`. Cette fonction doit être appelée lorsque l'utilisateur appuie sur un bouton "Yes" ou "No" et prend en paramètre d'entrée un booléen. Cependant il n'y a pour le moment aucune liaison entre une action réalisée sur la vue et le contrôle. Ainsi on crée de la même manière que pour les `@IBOutlet` ("control" + drag & drop) des `@IBAction` qui correspondent aux actions réalisées sur l'écran.

```
@IBAction func didTapYes(_ sender: Any) {  
    answerQuestion(answer: true)  
}  
  
@IBAction func didTapNo(_ sender: Any) {  
    answerQuestion(answer: false)  
}
```

Figure 20 : @IBActions utilisés dans l'application

### 3.1.5 Conclusion sur l'implémentation du modèle

*"Apple a choisi par défaut le très populaire patron de conception MVC pour les applications iPhone. Cela veut dire que la façon dont ils ont conçu le développement d'application iPhone nous encourage à respecter le design MVC", d'après la formation que j'ai suivie sur OpenClassrooms.*

Dans le cas de notre jeu simple, il y a peu de fichiers, les interactions Modèle/Controller et Controller/View sont peu nombreuses. On pourrait ajouter de nombreuses modifications aux projets et c'est là qu'est tout l'intérêt du modèle : il permet sans difficulté l'ajout ou la modification de fonctionnalités par un développeur (même extérieur au projet).

Il existe cependant des architectures d'applications plus élaborées pour des projets de plus grande ampleur.

## 3.2 Design Pattern 2 : Le Model View ViewModel

Pour présenter ce **modèle MVVM** j'ai choisi de créer une application qui calcule les crédits en fonction des matières validées. J'ai continué d'utiliser la bibliothèque UIKit pour les mêmes raisons que pour le précédent modèle. L'application est une Single View App pour permettre d'expliquer sa conception simplement.

Une nouvelle fois, je présente la manière que j'ai utilisé pour développer en utilisant le modèle MVVM mais elle n'est pas universelle.

### 3.2.1 Création de l'application

Le but de l'application est d'informer l'utilisateur du total de ses crédits validés et de lui dire quand il en a assez pour valider un diplôme. Pour présenter les différentes matières j'ai décidé d'utiliser un **TableView**. C'est une composante très utilisée avec UIKit qui permet de créer un tableau dynamique. Cela permet d'appliquer simplement le modèle MVVM et de modifier les données (dans notre cas des UEs) sans problèmes d'affichage.

Il y a une première page d'accueil qui mène vers notre page principale. La page principale contient le composant **UITableView** sur la partie supérieure, c'est celle-ci qui affichera les différentes UEs et avec laquelle l'utilisateur va interagir. On ne peut pas les faire apparaître sur le storyboard car le tableau dépend des données. Sur la partie inférieure il y a la somme des crédits dans chaque catégorie ainsi qu'un label qui indique si l'utilisateur a assez de crédit. La figure 21 montre l'ensemble des composants visibles par l'utilisateur.



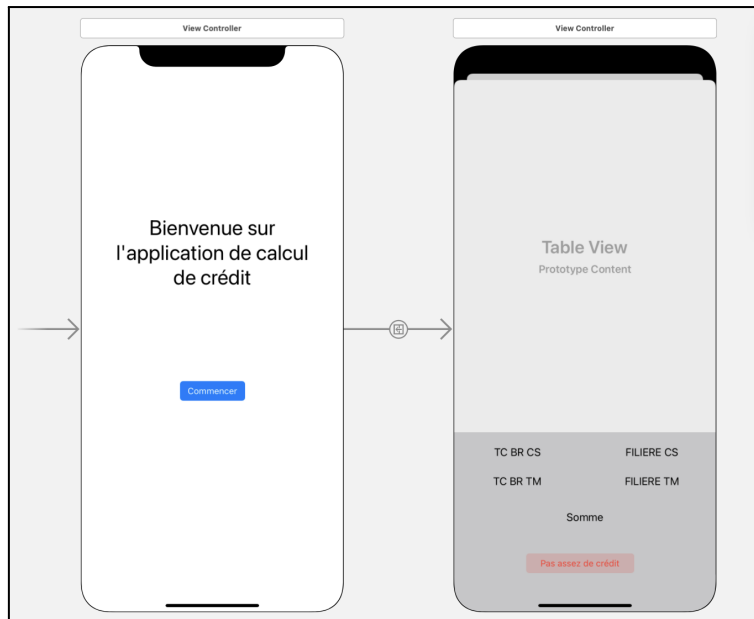
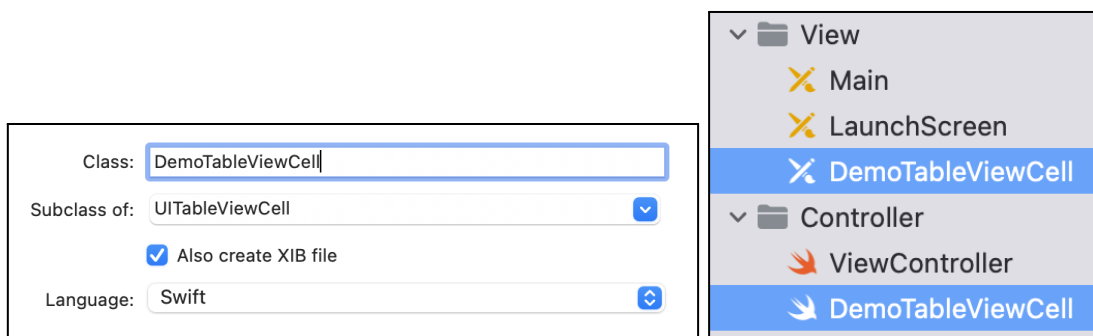


Figure 21 : Storyboard de l'application

Nous avons désormais toute la structure de l'application et il faut construire le tableau dynamique. Pour cela, pour rester dans la continuité du modèle MVVM j'ai choisi de diviser le code en différents fichiers. Pour cela, lorsqu'on utilise une TableView, on peut construire séparément du code principale les cellules. Ainsi, j'ai créé un fichier **DemoTableViewCell.swift** correspondant au controller d'une vue (et son fichier .xib associé qui correspond à la vue).



Figures 22 et 23 : Création d'une cellule dans un tableau

De la même manière que pour les fichiers **main.storyboard** et **ViewController.swift**, j'ai créé une vue avec des Outlets. Le but est d'avoir le nom de l'UE à gauche de la cellule, un bouton "+" permettant d'afficher l'intitulé de l'UE ainsi qu'un **UISwitch** qui indiquerait à l'application si l'UE est validée ou non (voir figure 24).



Figure 24 : Vue d'une cellule individuelle

Pour lier cette cellule au **UITableView** il faut créer un nouvel objet **UINib** et l'enregistrer avec les instructions de la figure 25 lorsque la vue est chargée (donc dans la fonction `viewDidLoad`).

```
// Association de la vue imbriquée à la TableView
let nib = UINib(nibName: "DemoTableViewCell", bundle: nil)
tableView.register(nib, forCellReuseIdentifier: "DemoTableViewCell")
```

Figure 25 : Association de la vue imbriquée à la TableView

On a donc divisé le code de nos vues en plusieurs fichiers distincts, on va désormais voir comment le **ViewModel** permet de faire de même pour les données.

### 3.2.2 Implémentation du ViewModel

Tout d'abord nous utilisons deux modèles dans cette application. Un modèle **UE** qui permet de stocker toutes les informations sur une UE et un modèle **Cursus** qui permet de stocker le nombre de crédit dans chaque catégorie. Leurs données sont renseignées dans le ViewController. Voici leurs attributs :

<pre>struct UE {     // Structure définissant une UE     let Name : String     let Label : String     let Categorie : String     let creditNumber : Int     let Niveau : String }</pre>	<pre>struct Cursus {     // Structure définissant un Cursus     var creditsCS_TCBR : Int = 0     var creditsTM_TCBR : Int = 0     var creditsCS_FILIERE : Int = 0     var creditsTM_FILIERE : Int = 0      func somme() -&gt; Int {         // Calcul du total des crédits         return creditsCS_TCBR+creditsTM_TCBR+creditsCS_FILIERE+creditsTM_FILIERE     } }</pre>
---	---

Figure 26 et 27 : Structures définissant les modèles

Cependant l'utilisateur n'a pas besoin de connaître la catégorie de l'UE (CS/TM) ni si elle est proposée en TCBR ou en Filière. Il a juste besoin de voir le nom de l'UE et son Label pour la sélectionner ou non. Le ViewModel va donc nous servir ici à n'utiliser que les données nécessaires.

Pour cela on crée une nouvelle classe **CellViewModel** qui ne contient que ces deux attributs (Name + Label). On peut alors dans le Controller associer les données nécessaires à une nouvelle constante **viewModel** lors de la construction du tableau avec les instructions ci-dessous.

```
let model = data[indexPath.row] // Indique la donnée en fonction de la ligne
let viewModel = CellViewModel(Name: model.Name, Label: model.Label) // Créé une constante avec uniquement les données utiles
cell.myLabel.text = "\(viewModel.Name)" // Met à jour le nom de l'UE sur le Label de la cellule correspondante
```

Figure 28 : Utilisation du ViewModel dans le Controller

À présent nous avons une application avec une interface prête à être utilisée et il nous reste à la faire fonctionner.

### 3.2.3 Fonctionnement de l'application

L'utilisateur a deux possibilités pour interagir avec l'application : il peut appuyer sur le bouton plus pour voir le label de l'application et fixer la valeur d'un Switch sur on ou Off. Il y aura donc deux fonctions nécessaires pour gérer ces interactions : **didChangeSwitch()** et **didTapButton()**. Elles sont toutes les deux définies dans le fichier `ViewController.swift`

La fonction **didTapButton()** va vérifier si le titre du bouton est "+" et dans ce cas le transformer en un Label donnant l'intitulé de l'UE. La fonction prend en entrée un `UIButton`. Voici, ci-dessous, ce qu'elle permet de faire (figure 29).

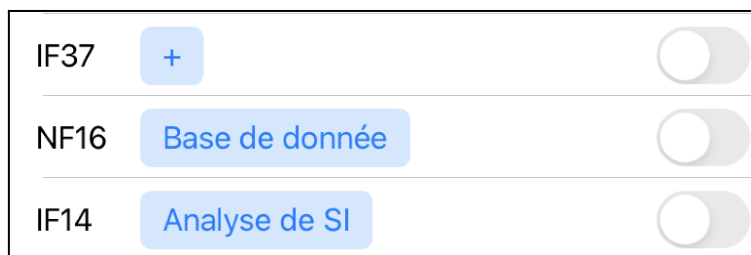


Figure 29 : Résultat d'interaction avec le bouton "+"

La fonction **didChangeSwitch()** est la fonction la plus importante de l'application. Elle permet de récolter la valeur d'un Switch en fonction de la cellule dans laquelle on se trouve. Par exemple, si on coche NF16, grâce à la fonction, l'application saura qu'il s'agit d'une CS de TCBR et incrémentera le nombre de crédits de cette catégorie. Cette fonction prend donc un `UISwitch` en paramètre d'entrée.

Ces fonctions sont appelées lorsque des événements apparaissent : pour un toucher avec le Button ou lorsque la valeur change pour le Switch. Ces événements sont définis lors de la création du tableau (voir figure 30). On voit bien à travers cet exemple que la bibliothèque **UIKit** est **EventDriven** (cf partie [2.1.4](#)).

```
cell.myButton.tag = indexPath.row
cell.myButton.addTarget(self, action: #selector(didTapButton(_)), for: .touchUpInside)
cell.mySwitch.tag = indexPath.row
cell.mySwitch.addTarget(self, action: #selector(didChangeSwitch(_)), for: .valueChanged)
```

Figure 30 : Utilisation des fonctions dans les cellules

Une fois les fonctions implémentées nous avons une application fonctionnelle qui répond aux besoins de l'utilisateur. La validation du diplôme demande par défaut 18 crédits pour simplifier les tests mais il est possible de modifier simplement ce nombre dans le code.



Figure 31 : Résultat du fonctionnement de l'application

### 3.2.4 Conclusion sur l'implémentation du modèle

J'ai trouvé que le modèle MVVM était plus dur à implémenter que le modèle MVC car la séparation des données est moins intuitive. Je pense qu'il est possible d'améliorer son utilisation dans cette application. J'ai aussi eu des difficultés pour gérer le passage des données entre les cellules du tableau et les fonctions pour que chaque Switch soit relié à la bonne UE.

Cependant le modèle MVVM est un modèle qui s'applique totalement à ce genre d'applications dans lesquelles toutes les données ne sont pas nécessaires à l'utilisateur. Le MVVM n'aurait pas été pertinent sur la première application (Jeu du nombre premier) par exemple mais s'applique bien à l'utilisation des **UITableView**.

## 3.3 Synthèse et point de vue

Les modèles MVC et MVVM sont, selon moi, deux bons modèles applicables au développement mobile. Ils sont robustes, relativement simples à mettre en place et adaptables sur une majorité de projets.

Le modèle MVC est le modèle le plus simple à utiliser. Apple dans le développement d'XCode et de son langage nous pousse vers l'utilisation du modèle grâce aux fichiers créés au lancement d'un projet et à la manière d'interagir entre les vues (le storyboard) et le Controller. Le modèle MVVM quant à lui est une version plus élaborée du MVC qui permet de mieux partitionner le code en isolant les données nécessaires et en créant des sous vues.

D'après mon expérience durant ce Projet Étudiant, je pense que le MVC tend à être utilisé lors de la réalisation d'applications de petite envergure. Par exemple des **Single View App** avec des fonctionnalités limitées. Le modèle MVVM est plus applicable à des moyennes et grosses applications. Par exemple, pour des applications avec des modèles très complets comportant beaucoup de données, le MVVM permettrait aux Controller des vues de ne récolter que données affichées par la suite, ce n'est pas le cas en MVC. On pourrait alors supposer que sur des applications complexes, le MVVM permet d'optimiser les ressources et d'augmenter la rapidité.

## 4 – Conclusion

### 4.1 Les missions et contraintes

Le but de ce Projet Étudiant était de découvrir et d'étudier l'environnement de développement mobile d'Apple à travers l'utilisation des Design Patterns. Il a fallu dans un premier temps découvrir l'environnement de développement Apple (IDE, syntaxe de Swift) avant d'ensuite réfléchir à la manière de coder les applications à l'aide des patrons de conceptions.

### 4.2 Les contributions

Pour **développer des applications mobiles sûres, viables et bien codées**, il est donc essentiel de disposer d'une boîte à outils de Design Patterns applicables en fonction du projet que l'on souhaite développer.

Pour cela j'ai étudié deux Design Patterns parmi les plus populaires sur Swift qui sont les modèles MVC et MVVM. Afin de comprendre au mieux leur fonctionnement j'ai réalisé deux applications : le Jeu du nombre et le Calcul de crédits. Ces deux applications m'ont permis de me faire mon propre avis sur les avantages et les inconvénients de chacun des modèles ainsi que sur leur applicabilité.

### 4.3 Ouverture du sujet

J'ai étudié deux Design Patterns durant ce Projet Étudiant mais il en existe de nombreux autres : MVP, Viper, Singleton... Il pourrait être intéressant d'aussi les étudier pour pouvoir les comparer avec les deux premiers.

D'autre part, dans ce projet étudiant j'ai réalisé toutes les contributions sur un émulateur d'iPhone 11. Cependant si on lance une même application sur des modèles d'iPhone différents, il est probable que l'application n'ai pas le rendu attendu. En effet, en fonction des tailles d'écran et des rapports longueurs/largeurs les applications peuvent être modifiées involontairement.

Pour pallier cela, il faut rendre l'application **responsive**, ce qui permet d'adapter le contenu de l'interface à tous les modèles. Swift permet très bien de le faire et il serait intéressant d'approfondir le projet en ce sens.


# Bibliographie

## 2 – Environnement de développement Apple


1. Formation Swift Openclassrooms : [Découvrez le rêve de Joe – Apprenez les fondamentaux de Swift – OpenClassrooms](#)
2. Formation Apple Swift UI : [SwiftUI Tutorials | Apple Developer Documentation](#)
3. Design Patterns by Tutorials de Jay Strawn & Joshua Green : [Design Patterns by Tutorials | Kodeco, the new raywenderlich.com](#)
4. Documentation Apple UIKit : [UIKit | Apple Developer Documentation](#)
5. Swift (langage d'Apple) [https://fr.wikipedia.org/wiki/Swift\\_\(langage\\_d%27Apple\)](https://fr.wikipedia.org/wiki/Swift_(langage_d%27Apple))

## 3 – Contributions

### 3.1 Design Pattern 1 : Model View Controller MVC

1. Formation Application Swift Openclassrooms : [Développez votre première application pour iOS – OpenClassrooms](#)
2. Formation MVC Openclassrooms : [Développez une application iPhone avec le modèle MVC – OpenClassrooms](#)
3. Vidéo Youtube MVC de Emmanuel Okwara  
 iOS Dev 33: MVC Design Pattern Explained with Example | Swift 5, XCode 13

### 3.2 Design Pattern 2 : Model View ViewModel

1. Vidéo Youtube MVVM de Emmanuel Okwara  
 iOS Dev 34: MVVM Design Pattern Explained with Example | Swift 5, XCode...
2. Tutoriel Développez.com : [Tutoriel pour apprendre à implémenter l'architecture MVVM sur iOS](#)
3. Tutoriel John Codeos : [How to implement MVVM pattern with Swift in iOS | John Codeos – Blog with Free iOS & Android Development Tutorials](#)
4. Formation Kodeco : [iOS MVVM Tutorial: Refactoring from MVC | Kodeco, the new raywenderlich.com](#)

## Annexes

Annexe 1 : TD IF26 iOS1 mis à jour



## Annexe 2 : Synthèse de syntaxe : correction TD IF26