UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Swift Combine

Utopios® Tous droits réservés

Programme

1. RxSwift

- Les Observables
- Les subjects
- Les Filtering operators
- Les Transforming Operators

Utopios® Tous droits réservés

Programme

2. Introduction à Combine

- Les éditeurs (Publishers) et les abonnés (Subscribers)
- Les opérateurs de transformation de flux
- La gestion des erreurs avec "error handling"
- Utilisation de "Schedulers"

Utopios® Tous droits réservés

Programme

3. RxSwift VS Combine
4. Introduction aux architectures iOS
    - Les différents modèles d'architecture iOS (MVC, MVVM, MVI, VIPER, etc.)
    - Les avantages et les inconvénients de chaque modèle

- Comment choisir le modèle d'architecture approprié pour votre projet

Utopios® Tous droits réservés

Programme

5. Flow Coordinator pattern pour la Navigation dans SwiftUI et StoryBoard

- Les avantages de l'utilisation du pattern Flow Coordinator pour la navigation
- Implémentation du pattern Flow Coordinator dans SwiftUI
- Implémentation du pattern Flow Coordinator dans Storyboard
- Bonnes pratiques
- Gestion la navigation complexe avec plusieurs Flows Coordinators imbriqués

Utopios® Tous droits réservés

Programme

6. Architecture MVI

- Présentation de l'architecture Model-View-Intent (MVI)
- Les rôles et les responsabilités de chaque élément de l'architecture
- Les avantages et les inconvénients de l'architecture MVI
- Les intentions (Intents) et leur rôle dans l'architecture MVI
- Les interactions entre les différentes parties de l'architecture MVI (Model, View, Intent)
- Utilisation de Combine pour la communication entre les différentes parties de l'architecture MVI

Utopios® Tous droits réservés

Programme

7. Modèle MVVM :

- Présentation de l'architecture Modèle-Vue-VueModèle (MVVM)
- Les rôles et les responsabilités de chaque élément de l'architecture
- Les avantages et les inconvénients de l'architecture MVVM
- Utilisation de Combine pour la communication entre la Vue et le VueModèle

Utopios® Tous droits réservés

Programme

8. Modèle VIPER :

- Présentation de l'architecture VIPER
- Les rôles et les responsabilités de chaque élément de l'architecture
- Les avantages et les inconvénients de l'architecture VIPER
- Utilisation de Combine pour la communication entre les différents éléments de l'architecture

Utopios® Tous droits réservés

Programme

9. Utilisation d'architectures combinées :

- Combinaison de plusieurs modèles d'architecture (développement modulaire)
- Les avantages et les inconvénients de l'utilisation de plusieurs modèles d'architecture

Utopios® Tous droits réservés

Programme

10. Gestion de l'état de l'application :

- Utilisation de Combine pour gérer l'état global de l'application
- Utilisation de modèles d'architecture pour gérer l'état local de chaque module de l'application

Utopios® Tous droits réservés

Programme

11. Tests unitaires et Développement par les TDD et BDD :

- Comment tester les différents éléments des modèles d'architecture
- Comment utiliser Combine pour tester les flux de données
- Comment implémenter le TDD avec Combine
- Les avantages de l'utilisation de Combine avec le TDD
- Utilisation de la bibliothèque Nimble pour faciliter l'écriture des tests unitaires
- Comment écrire des tests d'intégration pour le code qui utilise Combine
- Utilisation de l'opérateur "sink" pour obtenir les résultats de la combinaison des éditeurs
- Utilisation de l'opérateur "record" pour enregistrer et rejouer des flux de données dans les tests
- BDD (Behavior Driven Development) avec Swift et Combine
- Les scénarios dans BDD
- Les trois étapes du BDD (Given-When-Then)
- Comment implémenter les scénarios en utilisant Swift et Combine

Utopios® Tous droits réservés

RxSwift

Les Observables

Dans RxSwift, un Observable est un type qui représente une séquence d'événements. Un Observable peut émettre zéro, un ou plusieurs événements au fil du temps. Les observables sont le cœur de la programmation réactive, car ils permettent de créer des flux de données et de les manipuler facilement en chaînant des opérations.

Utopios® Tous droits réservés

RxSwift

Les Observables

```
import RxSwift

// Créer un Observable
let numbersObservable: Observable<Int> = Observable.of(1, 2, 3, 4, 5)

// S'abonner à l'Observable et manipuler les données
let disposeBag = DisposeBag()

numbersObservable
    .map { number in
        return number * 2
    }
    .subscribe(onNext: { doubledNumber in
        print("Le nombre doublé est :", doubledNumber)
    }, onCompleted: {
        print("La séquence est terminée.")
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Observables

- just : Crée un Observable qui émet une seule valeur et se termine
- of : Crée un Observable à partir d'une liste de valeurs
- from : Crée un Observable à partir d'un tableau, d'un dictionnaire ou d'un ensemble
- empty : Crée un Observable qui ne produit aucun élément et se termine immédiatement.
- never : Crée un Observable qui ne produit aucun élément et ne se termine jamais
- error : Crée un Observable qui termine immédiatement avec une erreur
- interval : Crée un Observable qui émet des éléments à intervalles réguliers.
- timer : Crée un Observable qui émet une valeur après un délai spécifié
- range : Crée un Observable qui émet une séquence de valeurs dans une plage spécifiée
- create : Crée un Observable personnalisé en fournissant une fonction qui définit son comportement
- deferred : Crée un Observable qui ne génère la séquence qu'au moment de l'abonnement

Utopios® Tous droits réservés

RxSwift

Les Subjects

Dans RxSwift, les Subject sont à la fois des observables et des observateurs. Ils peuvent émettre des événements et également réagir aux événements qui leur sont envoyés. Il existe quatre types de subjects dans RxSwift: PublishSubject, BehaviorSubject, ReplaySubject et BehaviorRelay.

Utopios® Tous droits réservés

RxSwift

Les Subjects

- PublishSubject: Il émet uniquement les événements qui se produisent après qu'un observateur s'est abonné. Les événements précédents ne sont pas reçus par les nouveaux abonnés

```
import RxSwift

let publishSubject = PublishSubject<String>()

publishSubject.onNext("Hello") // Ne sera pas reçu par l'observateur

let disposeBag = DisposeBag()

publishSubject.subscribe(onNext: { value in
    print("PublishSubject:", value)
}).disposed(by: disposeBag)

publishSubject.onNext("World") // Affiche : PublishSubject: World
```

Utopios® Tous droits réservés

RxSwift

Les Subjects

- BehaviorSubject: Il émet l'événement le plus récent à tout nouvel observateur au moment de l'abonnement et tous les événements ultérieurs

```
import RxSwift

let behaviorSubject = BehaviorSubject<String>(value: "Initial")

behaviorSubject.onNext("Hello")

let disposeBag = DisposeBag()

behaviorSubject.subscribe(onNext: { value in
    print("BehaviorSubject:", value)
}).disposed(by: disposeBag)

behaviorSubject.onNext("World") // Affiche : BehaviorSubject: World
```

Utopios® Tous droits réservés

RxSwift

Les Subjects

- ReplaySubject: Il émet un nombre spécifié d'événements précédents aux nouveaux observateurs et tous les événements ultérieurs.

```
import RxSwift

let replaySubject = ReplaySubject<String>.create(bufferSize: 2)

replaySubject.onNext("Event 1")
replaySubject.onNext("Event 2")
replaySubject.onNext("Event 3")

let disposeBag = DisposeBag()

replaySubject.subscribe(onNext: { value in
    print("ReplaySubject:", value)
}).disposed(by: disposeBag)

replaySubject.onNext("Event 4") // Affiche : ReplaySubject: Event 4
```

Utopios® Tous droits réservés

RxSwift

Les Subjects

- BehaviorRelay: Il ne permet pas d'émettre des erreurs ou de compléter l'événement. Il est utilisé pour stocker l'état dans les modèles de vue.

```
import RxCocoa
import RxSwift

let behaviorRelay = BehaviorRelay<String>(value: "Initial")

behaviorRelay.accept("Hello")

let disposeBag = DisposeBag()

behaviorRelay.subscribe(onNext: { value in
    print("BehaviorRelay:", value)
}).disposed(by: disposeBag)

behaviorRelay.accept("World") // Affiche : BehaviorRelay: World
```

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

Les opérateurs de filtrage dans RxSwift sont utilisés pour sélectionner ou filtrer les éléments émis par un Observable en fonction de certaines conditions. Ils permettent de transformer ou de filtrer les séquences en fonction de vos besoins

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

- filter: Cet opérateur émet uniquement les éléments qui correspondent à une condition spécifiée

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .filter { $0 % 2 == 0 }
    .subscribe(onNext: { value in
        print("Filter:", value) // Affiche uniquement les nombres pairs
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

- distinctUntilChanged: Cet opérateur supprime les éléments consécutifs identiques

```
import RxSwift

let disposeBag = DisposeBag()
let values = Observable.of(1, 1, 2, 2, 3, 3, 4, 4)

values
    .distinctUntilChanged()
    .subscribe(onNext: { value in
        print("DistinctUntilChanged:", value) // Affiche : 1, 2, 3, 4
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

- take: Cet opérateur prend un nombre spécifié d'éléments à partir du début de la séquence

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .take(3)
    .subscribe(onNext: { value in
        print("Take:", value) // Affiche : 1, 2, 3
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

- skip: Cet opérateur ignore un nombre spécifié d'éléments au début de la séquence

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .skip(2)
    .subscribe(onNext: { value in
        print("Skip:", value) // Affiche : 3, 4, 5
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Filtering operators

- takeWhile: Cet opérateur prend les éléments tant qu'ils correspondent à une condition spécifiée

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .takeWhile { $0 < 4 }
    .subscribe(onNext: { value in
        print("TakeWhile:", value) // Affiche : 1, 2, 3
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

Les opérateurs de transformation (Transforming Operators) dans RxSwift sont utilisés pour transformer ou modifier les événements émis par un observable. Ils permettent de changer la forme ou la structure des événements pour mieux s'adapter à vos besoins.

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

- map : Transforme chaque événement en appliquant une fonction de transformation

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .map { number in
        return number * 2
    }
    .subscribe(onNext: { doubledNumber in
        print("Transforming Operator - Map:", doubledNumber)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - Map: 2
//         Transforming Operator - Map: 4
//         Transforming Operator - Map: 6
//         Transforming Operator - Map: 8
//         Transforming Operator - Map: 10
```

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

- flatMap : Transforme chaque événement en un observable et fusionne les observables résultants en une seule séquence

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .flatMap { number in
        return Observable.of(number, number * 2, number * 3)
    }
    .subscribe(onNext: { transformedNumber in
        print("Transforming Operator - FlatMap:", transformedNumber)
    })
    .disposed(by: disposeBag)
```

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

- scan : Applique une opération à chaque événement et émet le résultat intermédiaire à chaque étape

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .scan(0) { accumulated, next in
        return accumulated + next
    }
    .subscribe(onNext: { accumulatedSum in
        print("Transforming Operator - Scan:", accumulatedSum)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - Scan: 1
//         Transforming Operator - Scan: 3
//         Transforming Operator - Scan: 6
//         Transforming Operator - Scan: 10
//         Transforming Operator - Scan: 15
```

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

- buffer : Rassemble les événements émis par l'observable dans des tableaux et les émet une fois que le tableau est plein

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
.buffer(timeSpan: .seconds(3), count: 2, scheduler: MainScheduler.instance)
.subscribe(onNext: { numbersArray in
print("Transforming Operator - Buffer:", numbersArray)
})
.disposed(by: disposeBag)

// Output: Transforming Operator - Buffer: [1, 2]
// Transforming Operator - Buffer: [3, 4]
// Transforming Operator - Buffer: [5]
```

Utopios® Tous droits réservés

RxSwift

Les Transforming Operators

- groupBy : Divise l'observable en sous-observables en fonction d'une clé.

```
import RxSwift

let wordsObservable = Observable.of("apple", "banana", "cherry", "avocado", "blueberry")

wordsObservable
    .groupBy { word in
        return word.count
    }
    .flatMap { group in
        return group.toArray()
    }
    .subscribe(onNext: { wordsArray in
        print("Transforming Operator - GroupBy:", wordsArray)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - GroupBy: ["apple"]
```

Utopios® Tous droits réservés

Introduction à Combine

- Le framework Combine est basé sur le concept de programmation réactive. Il permet de manipuler des flux de données, appelés publishers, qui émettent des événements, tels que des valeurs, des erreurs ou la fin de la séquence. Ces publishers peuvent être modifiés à l'aide d'opérateurs, tels que map, filter ou reduce, pour produire des valeurs transformées.
- Les valeurs produites peuvent ensuite être transmises à des abonnés, qui peuvent s'inscrire aux publishers à l'aide de la méthode sink. Les abonnés peuvent également être des publishers eux-mêmes, permettant la création de chaînes complexes de flux de données.
- En plus des publishers et des abonnés, le framework Combine fournit également une variété d'autres types pour aider à gérer les flux de données, tels que Subject, qui permet de créer des publishers personnalisés, et Cancellable, qui permet d'annuler des abonnements.
- Le framework Combine est très utile pour les tâches asynchrones dans les applications iOS/macOS, telles que les appels réseau, les accès à la base de données ou les événements d'interface utilisateur. Il permet de gérer de manière élégante les flux de données asynchrones et de les traiter de manière déclarative, ce qui peut rendre le code plus clair, plus concis et plus facile à maintenir.

Utopios® Tous droits réservés

Combine

Publishers (éditeurs)

- les Publishers (éditeurs) sont des objets qui émettent des valeurs au fil du temps
- Les Publishers dans Combine peuvent être de plusieurs types, tels que les publishers de données (Data Publishers), les publishers de notifications (Notification Publishers), les publishers de temporisation (Timer Publishers) et les publishers personnalisés (Custom Publishers)

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

numbersPublisher
    .map { $0 * 2 }
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Subscribers (abonnés)

- Les Subscribers sont des objets qui reçoivent et traitent les valeurs émises par les publishers. Les abonnés sont créés à l'aide de la méthode sink, qui permet de spécifier une ou plusieurs closures pour traiter les valeurs produites.

```
import Combine
let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

let numbersSubscriber = Subscribers.Sink<Int, Never>(receiveCompletion: {e in print(e)},receiveValue: { value in
            print(value)
        })

numbersPublisher.subscribe(numbersSubscriber)
```

Utopios® Tous droits réservés

Combine

Publishers - Publishers de données

- Publishers de données (Data Publishers) : Ce type de publisher émet une séquence de valeurs, telles que des nombres, des chaînes de caractères, des tableaux, etc. Les publishers de données sont créés à l'aide de la méthode publisher() ou de la méthode Just().

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

numbersPublisher
    .map { $0 * 2 }
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Publishers - Publishers de notifications

- Publishers de notifications (Notification Publishers) : Ce type de publisher émet des notifications, telles que des notifications de fin de vie d'application, de changement de langue, etc. Les publishers de notifications sont créés à l'aide des notifications NotificationCenter.

```
import Combine

let notificationPublisher = NotificationCenter.default.publisher(for: UIApplication.didEnterBackgroundNotification)

notificationPublisher
    .sink { notification in
        print("Application did enter background")
    }
```

Utopios® Tous droits réservés

Combine

Publishers - Publishers de temporisation

- Publishers de temporisation (Timer Publishers) : Ce type de publisher émet des événements à des intervalles réguliers, tels que des timers. Les publishers de temporisation sont créés à l'aide de la méthode Timer.publish().

```
import Combine

let timerPublisher = Timer.publish(every: 1.0, on: .main, in: .common)

timerPublisher
    .sink { _ in
        print("Timer fired")
    }
```

Utopios® Tous droits réservés

Combine

Publishers - Publishers personnalisés

- Publishers personnalisés (Custom Publishers) : Ce type de publisher permet de créer des publishers personnalisés en conformité avec le protocole Publisher. Les publishers personnalisés peuvent être créés pour émettre des valeurs spécifiques ou pour se connecter à des sources de données personnalisées.

```
import Combine

class MyCustomPublisher: Publisher {
    typealias Output = String
    typealias Failure = Never

    func receive<S>(subscriber: S) where S : Subscriber, Failure == S.Failure, Output == S.Input {
        subscriber.receive("Hello, world!")
        subscriber.receive(completion: .finished)
    }
}
let customPublisher = MyCustomPublisher()

customPublisher
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Subscribers - Sink

- Sink : Le subscriber Sink est utilisé pour recevoir des valeurs émises par un publisher, et effectuer une ou plusieurs opérations sur ces valeurs.

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

let numbersSubscriber = Subscribers.Sink<Int, Never>(receiveValue: { value in
    print(value)
})

numbersPublisher.subscribe(numbersSubscriber)
```

Utopios® Tous droits réservés

Combine

Subscribers - Assign

- Assign : Le subscriber Assign est utilisé pour assigner les valeurs émises par un publisher à une propriété d'un objet.

```
import Combine

class MyClass {
    var myValue: Int = 0
}

let myClass = MyClass()
let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

numbersPublisher.assign(to: \.myValue, on: myClass)
```

Utopios® Tous droits réservés

Combine

Subscribers - AnySubscriber

- AnySubscriber : Le subscriber AnySubscriber permet de créer un subscriber générique pour recevoir des valeurs de n'importe quel type.

```
import Combine

let numbersPublisher =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

let anySubscriber = AnySubscriber<Int, Never>(
    receiveSubscription: { subscription in
        subscription.request(.unlimited)
    },
    receiveValue: { value -> Subscribers.Demand in
        print(value)
        return .unlimited
    },
    receiveCompletion: { completion in
        print(completion)
    }
)
numbersPublisher.subscribe(anySubscriber)
```

Utopios® Tous droits réservés

Combine

Subscribers - PassthroughSubject

- PassthroughSubject : Le subscriber PassthroughSubject est utilisé pour créer un publisher personnalisé qui émet des valeurs, et pour recevoir des valeurs émises par ce publisher.

```
import Combine

let passthroughSubject = PassthroughSubject<String, Never>()

passthroughSubject
    .sink { value in
        print("Received value: \(value)")
    }

passthroughSubject.send("Hello, world!")
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

Les opérateurs de transformation de flux dans Combine sont des fonctions qui prennent un ou plusieurs éditeurs (Publishers) en entrée et renvoient un nouvel éditeur en sortie. Ces opérateurs transforment les flux d'événements en appliquant des opérations telles que le filtrage, le mappage, le regroupement et la combinaison. Voici quelques-uns des opérateurs de transformation de flux les plus couramment

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- map: L'opérateur map transforme chaque élément de l'éditeur en appliquant une fonction à l'élément. Il renvoie un nouvel éditeur avec les éléments transformés

```
let publisher =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])
let mappedPublisher = publisher.map { $0 * 2 }
mappedPublisher.sink { value in
    print(value) // Affiche 2, 4, 6, 8, 10
}
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- filter: L'opérateur filter élimine certains éléments de l'éditeur en appliquant une fonction de filtrage à chaque élément. Il renvoie un nouvel éditeur contenant uniquement les éléments qui satisfont la condition de filtrage.

```
let publisher =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])
let filteredPublisher = publisher.filter { $0 % 2 == 0 }
filteredPublisher.sink { value in
    print(value) // Affiche 2, 4
}
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- flatMap: L'opérateur flatMap transforme chaque élément de l'éditeur en un ou plusieurs nouveaux éditeurs en appliquant une fonction à l'élément. Les éditeurs générés sont ensuite combinés en un seul éditeur. Il renvoie un nouvel éditeur contenant les éléments combinés.

```
let publisher =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])
let flatMapPublisher = publisher.flatMap { value in
    return [value, value * 2, value * 3].publisher
}
flatMapPublisher.sink { value in
    print(value) // Affiche 1, 2, 3, 2, 4, 6, 3, 6, 9, 4, 8, 12, 5, 10, 15
}
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- scan: L'opérateur scan combine les éléments successifs de l'éditeur en appliquant une fonction d'accumulateur. Il renvoie un nouvel éditeur qui émet les résultats intermédiaires de l'accumulateur.

```
let publisher =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])
let scanPublisher = publisher.scan(0) { accumulator, value in
    return accumulator + value
}
scanPublisher.sink { value in
    print(value) // Affiche 1, 3, 6, 10, 15
}
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- merge: L'opérateur merge combine plusieurs éditeurs en un seul éditeur qui émet des éléments de tous les éditeurs sources. Il renvoie un nouvel éditeur qui combine les éléments de tous les éditeurs sources.

```
let publisher1 =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3])
let publisher2 =  Publishers.Sequence<[Int], Never>(sequence: [4,5,6])
let mergedPublisher = Publishers.Merge(publisher1, publisher2)
mergedPublisher.sink { value in
    print(value) // Affiche 1, 2, 3, 4, 5, 6
}
```

Utopios® Tous droits réservés

Combine

Les opérateurs de transformation de flux

- combineLatest: L'opérateur combineLatest combine les derniers éléments de plusieurs éditeurs en appliquant une fonction à ces éléments. Il renvoie un nouvel éditeur qui émet les résultats de la fonction appliquée aux derniers éléments de tous les éditeurs.

```
let publisher1 =  Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3])
let publisher2 =  Publishers.Sequence<[Int], Never>(sequence: [4,5,6])
let combinedPublisher = Publishers.CombineLatest(publisher1, publisher2)
    .map { $0 + $1 }
combinedPublisher.sink { value in
    print(value) // Affiche 5, 6, 7, 6, 7, 8, 7, 8, 9
}
```

Utopios® Tous droits réservés

Combine

La gestion des erreurs

La gestion des erreurs (error handling) dans le framework Combine permet de gérer les erreurs émises par les publishers lors de la production des valeurs. Les erreurs peuvent être générées par des erreurs de réseau, des erreurs d'accès à la base de données, des erreurs de parsing, etc. Pour gérer les erreurs, le framework Combine propose plusieurs opérateurs.

Utopios® Tous droits réservés

Combine

La gestion des erreurs

- catch: Cet opérateur permet de remplacer une erreur par une autre valeur ou par un nouveau publisher.

```
import Combine
enum CustomError: Error {
    case networkError
    case parsingError
}
let numbersPublisher =  Publishers.Sequence<[Int], Error>(sequence: [1, 2, 3, 4, 5])
numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.parsingError
        }
        return value
    }
    .catch { error -> Just<Int> in
        if let customError = error as? CustomError {
            switch customError {
            case .parsingError:
                return Just(0)
            default:
                return Just(1)
            }
        }
        return Just(2)
    }
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

La gestion des erreurs

- retry: Cet opérateur permet de retenter la production des valeurs en cas d'erreur.

```
import Combine
enum CustomError: Error {
    case networkError
}
var counter = 0
let numbersPublisher = PassthroughSubject<Int, CustomError>()
let subscription = numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.networkError
        }
        return value
    }
    .retry(2)
    .sink(
        receiveCompletion: { completion in
            print(completion)
        },
        receiveValue: { value in
            print(value)
        }
    )
numbersPublisher.send(1)
numbersPublisher.send(2)
numbersPublisher.send(3)
```

Utopios® Tous droits réservés

Combine

La gestion des erreurs

- replaceError: Cet opérateur permet de remplacer toutes les erreurs par une autre valeur.

```
import Combine

enum CustomError: Error {
    case networkError
}

let numbersPublisher = Publishers.Sequence<[Int], Error>(sequence: [1, 2, 3, 4, 5])

numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.networkError
        }
        return value
    }
    .replaceError(with: 0)
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Schedulers

Les "schedulers" dans le framework Combine sont utilisés pour définir la façon dont les opérations sont exécutées, notamment pour définir l'ordonnancement des tâches et la file d'attente d'exécution. Le framework Combine fournit plusieurs schedulers pour les différentes tâches, comme le travail sur la file d'attente principale, l'exécution sur un thread spécifique ou un travail sur un thread background.

Utopios® Tous droits réservés

Combine

Schedulers

- MainScheduler : Ce scheduler est utilisé pour exécuter les opérations sur la file d'attente principale.

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

numbersPublisher
    .receive(on: DispatchQueue.main)
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Schedulers

- DispatchQueue : Ce scheduler est utilisé pour exécuter les opérations sur une file d'attente personnalisée.

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

let backgroundQueue = DispatchQueue(label: "my-background-queue")

numbersPublisher
    .subscribe(on: backgroundQueue)
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Schedulers

- OperationQueue : Ce scheduler est utilisé pour exécuter les opérations sur une file d'attente de type OperationQueue.

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Never>(sequence: [1, 2, 3, 4, 5])

let operationQueue = OperationQueue()

numbersPublisher
    .subscribe(on: operationQueue)
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

Schedulers

- ImmediateScheduler : Ce scheduler est utilisé pour exécuter les opérations immédiatement.

```
import Combine

let numbersPublisher = Publishers.Sequence<[Int], Error>(sequence: [1, 2, 3, 4, 5])

numbersPublisher
    .subscribe(on: ImmediateScheduler.shared)
    .sink { print($0) }
```

Utopios® Tous droits réservés

Combine

RxSwift VS Combine

1. Origine et plateformes supportées :

- RxSwift : Il s'agit d'une implémentation de ReactiveX (Reactive Extensions) pour Swift. RxSwift est une bibliothèque open-source développée par la communauté et est compatible avec iOS, macOS, watchOS et tvOS.
- Combine : Combine est un framework de programmation réactive développé par Apple et introduit à partir d'iOS 13, macOS 10.15, watchOS 6 et tvOS 13. Il est intégré directement dans les frameworks Apple et bénéficie du support officiel d'Apple.

Utopios® Tous droits réservés

Combine

RxSwift VS Combine

2. Compatibilité et intégration :

- RxSwift : Comme RxSwift est indépendant d'Apple, il fonctionne sur des versions antérieures d'iOS (iOS 8+), ce qui peut être important si vous devez prendre en charge des versions plus anciennes d'iOS. De plus, RxSwift dispose d'une vaste collection d'extensions pour les frameworks Apple, comme RxCocoa pour UIKit et AppKit.
- Combine : Combine nécessite iOS 13+ et les versions correspondantes pour les autres plateformes Apple. Il est étroitement intégré aux frameworks Apple et fonctionne parfaitement avec SwiftUI, ce qui permet une meilleure expérience de développement lorsque vous utilisez les technologies récentes d'Apple.

Utopios® Tous droits réservés

Combine

RxSwift VS Combine

3.Concepts et terminologie :

- RxSwift : RxSwift utilise des concepts et une terminologie similaires à ceux de ReactiveX, y compris des éléments tels que Observable, Observer, Subject, Single, Maybe, Completable, etc.
- Combine : Combine utilise une terminologie légèrement différente et simplifiée par rapport à ReactiveX. Par exemple, il utilise Publisher, Subscriber, Subject, Future, PassthroughSubject et CurrentValueSubject. Les concepts sont similaires, mais les noms diffèrent pour mieux correspondre aux conventions de Swift et d'Apple.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

Les architectures iOS sont des approches structurées pour organiser et diviser le code d'une application en composants modulaires et réutilisables. Elles favorisent la séparation des préoccupations, facilitent la maintenance du code et permettent de créer des applications évolutives et testables.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

1. MVC (Modèle-Vue-Contrôleur) :

- Le MVC est une architecture traditionnelle et le modèle standard pour les applications UIKit. Il divise l'application en trois composants principaux : Modèle, Vue et Contrôleur.
  - Modèle : Il représente les données et la logique métier de l'application.
  - Vue : Il s'agit de la représentation visuelle des données, principalement composée d'éléments d'interface utilisateur.
  - Contrôleur : Il fait le lien entre le modèle et la vue, gère les entrées utilisateur et les mises à jour de la vue en fonction des changements de données.
    Un problème courant avec MVC est le "Massive View Controller", où le contrôleur devient trop volumineux et difficile à maintenir. Il est important d'éviter de surcharger les contrôleurs et de répartir les responsabilités de manière appropriée.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

2. MVP (Modèle-Vue-Présentateur) :

- Le MVP est une évolution du MVC qui vise à améliorer la séparation des préoccupations et à faciliter les tests.
  - Modèle : Similaire au MVC, il représente les données et la logique métier de l'application.
  - Vue : Il s'agit de la représentation visuelle des données, mais elle est plus passive que dans le MVC et délègue la gestion des événements au présentateur.
  - Présentateur : Il gère les entrées utilisateur, les mises à jour de la vue et interagit avec le modèle pour récupérer ou modifier les données. La vue et le présentateur sont généralement définis par des protocoles, ce qui facilite la création de tests unitaires.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

3. MVVM (Modèle-Vue-Modèle de Vue) :

- Le MVVM est une autre évolution du MVC qui vise à améliorer la séparation des préoccupations et à faciliter les tests et l'intégration avec des frameworks de liaison de données, comme SwiftUI et Combine.
  - Modèle : Il représente les données et la logique métier de l'application.
  - Vue : Il s'agit de la représentation visuelle des données, généralement composée d'éléments d'interface utilisateur.
  - Modèle de Vue (ViewModel) : Il sert d'intermédiaire entre la vue et le modèle, exposant les données et les commandes nécessaires pour la vue. Le ViewModel n'a pas de référence directe à la vue, ce qui permet une plus grande réutilisabilité et des tests plus simples.
    MVVM est particulièrement adapté aux applications utilisant des frameworks de programmation réactive et de liaison de données, comme SwiftUI et Combine.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

3. bis Le MVI est une variante du modèle Model-View-ViewModel (MVVM) et se concentre sur le flux unidirectionnel des données et la gestion des actions de l'utilisateur sous forme d'intentions (Intents). Les composants clés du MVI sont :
    - Model : Il représente l'état de l'application et contient les données et la logique métier.
    - View : Il s'agit de la représentation visuelle des données, généralement composée d'éléments d'interface utilisateur. La vue observe les changements dans le modèle et met à jour l'affichage en conséquence.
    - Intent : Les intentions représentent les actions de l'utilisateur, telles que les clics de bouton, les saisies de texte, etc. Les intentions sont traitées et transformées en modifications d'état du modèle.

Utopios® Tous droits réservés

Combine

Introduction aux architectures iOS

4. VIPER (Vue-Interacteur-Présentateur-Entité-Routeur) :

- VIPER est une architecture basée sur le principe de la responsabilité unique et le modèle de conception Clean Architecture. Elle vise à créer des modules hautement découplés et testables.
  - Vue : Il s'agit de la représentation visuelle des données, généralement composée d'éléments d'interface utilisateur. La vue est passive et communique avec le présentateur pour les mises à jour et les actions de l'utilisateur.
  - Interacteur : Il contient la logique métier et les règles de l'application. L'interacteur récupère les données à partir de différentes sources et les prépare pour le présentateur.
  - Présentateur : Il gère les entrées utilisateur, les mises à jour de la vue et interagit avec l'interacteur pour récupérer ou modifier les données. Le présentateur est responsable de la logique de présentation, en transformant les données pour les afficher dans la vue.
  - Entité : Il s'agit des objets de modèle qui représentent les données dans l'application.
  - Routeur : Il gère la navigation entre les modules et est responsable de la création et de la configuration des modules VIPER.

Utopios® Tous droits réservés

Combine

Flow Coordinator pattern pour la Navigation dans SwiftUI et StoryBoard

Avantages

Le Flow Coordinator, également connu sous le nom de Coordinator pattern, est une approche pour la gestion de la navigation et du flux d'écrans dans une application iOS. L'utilisation du pattern Flow Coordinator présente plusieurs avantages :

1. Séparation des préoccupations : Le Flow Coordinator déplace la logique de navigation et la gestion des dépendances hors des contrôleurs de vue. Cela permet d'avoir des contrôleurs de vue plus légers et plus concentrés sur leur propre logique de présentation.
2. Réutilisabilité : Les contrôleurs de vue deviennent plus réutilisables, car ils ne sont pas liés à un flux de navigation spécifique. Vous pouvez réutiliser les mêmes contrôleurs de vue dans différents contextes de navigation sans modifier leur code interne.
3. Testabilité : Avec la logique de navigation et la gestion des dépendances séparées des contrôleurs de vue, il est plus facile d'écrire des tests unitaires pour ces éléments. Les Flow Coordinators peuvent être testés indépendamment des contrôleurs de vue et vice versa.

Utopios® Tous droits réservés

Combine

Flow Coordinator pattern pour la Navigation dans SwiftUI et StoryBoard

Avantages

4. Modularité : Le pattern Flow Coordinator encourage la modularité en permettant de diviser l'application en unités fonctionnelles indépendantes. Chaque module peut avoir son propre Flow Coordinator, ce qui facilite la maintenance et l'évolutivité de l'application.
5. Flexibilité : La gestion centralisée de la navigation permet de modifier facilement le flux de navigation sans avoir à toucher aux contrôleurs de vue. Vous pouvez ajouter, supprimer ou réorganiser des écrans sans impacter le reste de l'application.
6. Amélioration de la communication entre les écrans : Le Flow Coordinator peut gérer la communication entre les écrans en transmettant les données et en gérant les dépendances. Cela évite les couplages forts entre les contrôleurs de vue et rend l'ensemble du système plus flexible et plus facile à comprendre.

Utopios® Tous droits réservés

SwiftUI et Combine

Mise En place d'une Architecture MVI

- L'objectif est de créer une application de gestionnaire de tâches en utilisant SwiftUI, l'architecture MVI (Model-View-Intent) et le framework Combine
- Nous souhaitons réaliser une application qui permettra aux utilisateurs de créer, modifier et supprimer des tâches. Les utilisateurs pourront également marquer les tâches comme favorites. Nous utiliserons SwiftUI pour l'interface utilisateur, l'architecture MVI pour structurer votre code et Combine pour gérer les événements et les mises à jour d'état.

Utopios® Tous droits réservés

SwiftUI et Combine

Mise En place d'une Architecture MVI

1. Création d'un model Task avec les propriétés suivantes :
    - id : UUID unique pour chaque tâche.
    - title : Titre de la tâche (String).
    - isFavorite : Indique si la tâche est marquée comme favorite (Bool).
2. Implémentation l'architecture MVI en créant les éléments suivants :
    - TaskListState : Structure représentant l'état de l'écran de la liste des tâches.
    - TaskListEvent : Enumération des événements possibles (ajouter une tâche, mettre à jour une tâche, supprimer une tâche, basculer le statut de favori).
    - TaskListIntent : Classe qui traite les événements et met à jour l'état.

Utopios® Tous droits réservés

SwiftUI et Combine

Mise En place d'une Architecture MVI

3. Utilisation de SwiftUI pour créer les vues suivantes :
    - TaskListView : Vue principale affichant la liste des tâches avec la possibilité d'en ajouter de nouvelles. Utilisez NavigationLink pour naviguer vers la vue de détail des tâches lorsqu'une tâche est sélectionnée.
    - TaskDetailView : Vue de détail permettant de visualiser et modifier les détails d'une tâche, notamment son titre et son statut de favori. Incluez des boutons pour enregistrer les modifications ou annuler.
4. Utilisez Combine pour gérer la communication entre les vues et les intentions.
5. Les fonctionnalités sont :
    - Ajout d'une nouvelle tâche.
    - Modification du titre et du statut de favori d'une tâche existante.
    - Suppression d'une tâche.
    - Basculer le statut de favori d'une tâche.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Combinaison de plusieurs modèles d'architecture (développement modulaire)

La combinaison de plusieurs modèles d'architecture, également connue sous le nom de développement modulaire, est une approche de conception logicielle qui vise à tirer parti des avantages de différents modèles d'architecture pour créer une structure de code plus robuste, flexible et maintenable. Cette approche permet de mieux adapter le code aux besoins spécifiques d'un projet et d'améliorer la collaboration entre les membres de l'équipe.
Dans le contexte des applications développées avec SwiftUI et Combine, cette combinaison peut être particulièrement bénéfique pour gérer la complexité croissante et les défis de la création d'applications réactives et événementielles.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Les avantages et les inconvénients de l'utilisation de plusieurs modèles d'architecture

Voici quelques avantages clés de la combinaison de plusieurs modèles d'architecture :

- Flexibilité : En combinant des modèles d'architecture, vous pouvez créer une structure de code qui répond aux exigences spécifiques de votre projet, permettant une plus grande adaptabilité et évolutivité.
- Réutilisabilité : En utilisant une approche modulaire, vous pouvez créer des composants réutilisables pour différentes parties de votre application, ce qui facilite la maintenance et réduit les efforts de développement.
- Testabilité : En séparant clairement les responsabilités et les préoccupations, il est plus facile de créer des tests unitaires et d'intégration pour les différentes parties de votre application.
- Collaboration : Une architecture combinée et modulaire facilite la collaboration entre les membres de l'équipe, car chaque développeur peut se concentrer sur une partie spécifique de l'application sans interférer avec le travail des autres.
- Meilleure organisation du code : En combinant différents modèles d'architecture, vous pouvez organiser votre code de manière plus structurée, en évitant les dépendances excessives et en maintenant une séparation claire des responsabilités.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Les avantages et les inconvénients de l'utilisation de plusieurs modèles d'architecture

L'utilisation de plusieurs modèles d'architecture présente également certains inconvénients et défis. Voici quelques-uns des principaux inconvénients à prendre en compte :

1. Complexité accrue : La combinaison de différents modèles d'architecture peut entraîner une complexité accrue dans la structure de code, ce qui rend plus difficile la compréhension et la maintenance du code pour les développeurs.
2. Courbe d'apprentissage : Apprendre et maîtriser plusieurs modèles d'architecture peut demander un investissement en temps et en ressources pour les membres de l'équipe, en particulier pour ceux qui sont moins familiers avec certains modèles.
3. Intégration difficile : Il peut être difficile d'intégrer différents modèles d'architecture de manière cohérente, en particulier lorsqu'ils ont des principes et des méthodes de travail différents.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Les avantages et les inconvénients de l'utilisation de plusieurs modèles d'architecture

4. Risque de sur-conception : En essayant de combiner plusieurs modèles d'architecture, il y a un risque de sur-conception et de création d'une architecture trop complexe pour les besoins réels du projet, ce qui peut nuire à la productivité et à l'efficacité du développement.
5. Incohérence : Si l'équipe n'est pas bien alignée sur l'utilisation des différents modèles d'architecture, il peut y avoir des incohérences dans la manière dont les différentes parties du code sont structurées et gérées, ce qui peut entraîner des problèmes de maintenance et de collaboration.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Utilisation de l'injection de dépendances et IoC

L'injection de dépendance et l'inversion de contrôle sont des techniques de conception logicielle qui aident à créer des applications modulaires, testables et maintenables.

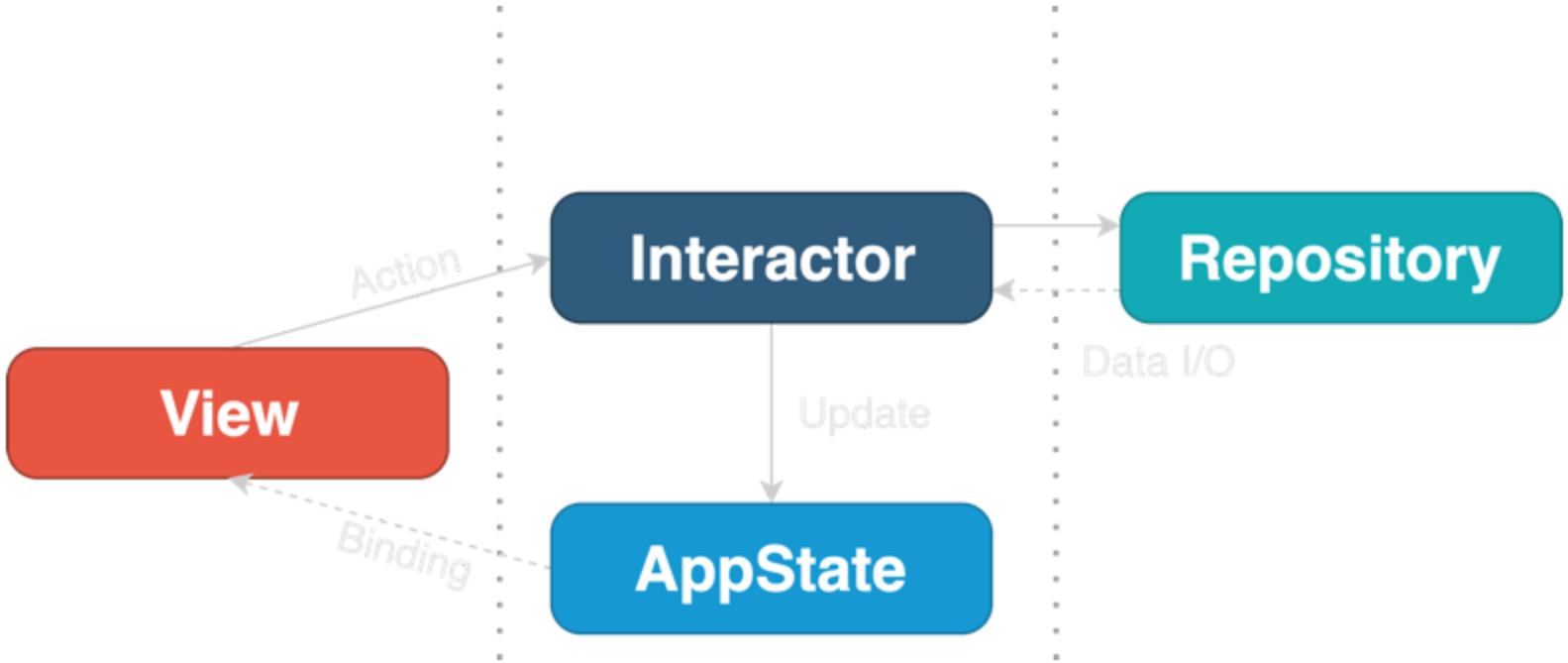
1. Créez des protocoles pour vos dépendances :
   Définissez des protocoles pour les services ou les objets dont vos composants SwiftUI ont besoin. Ces protocoles décrivent les interfaces pour les dépendances et permettent une inversion de contrôle.
2. Implémentez les classes qui respectent ces protocoles :
   Créez des classes concrètes qui implémentent les protocoles définis. Ces classes fournissent les fonctionnalités réelles pour les dépendances.
3. Utilisez l'environnement SwiftUI pour l'injection de dépendance :
   SwiftUI fournit un mécanisme appelé @EnvironmentObject qui permet d'injecter des dépendances dans vos composants de manière décentralisée. Vous pouvez créer des instances de vos classes concrètes et les injecter dans l'environnement de votre application.

Utopios® Tous droits réservés

Utilisation d'architectures combinées

Utilisation des interactor

Les interactors sont des composants qui gèrent la logique métier de votre application. Ils sont souvent utilisés dans le cadre de l'architecture Clean Swift (aussi appelée VIP) pour séparer les responsabilités entre la présentation, la logique métier et la récupération de données.



Utopios® Tous droits réservés

Gestion de l'état de l'application :

Utilisation de Combine pour gérer l'état global de l'application

L'AppState fait référence à l'état global de l'application, qui est souvent utilisé pour gérer les données et les informations partagées entre différentes parties de l'application. L'utilisation d'AppState dans le cadre de l'architecture Clean Swift avec SwiftUI présente plusieurs avantages.

1. Séparation des préoccupations: En utilisant AppState pour gérer l'état global de l'application, vous pouvez séparer la logique métier de la logique de présentation. Cela rend votre code plus facile à comprendre, à maintenir et à tester.
2. Partage de données: AppState permet de partager facilement des données entre différentes parties de l'application sans avoir à passer explicitement les données entre les vues. Cela simplifie la communication entre les composants et permet d'éviter les problèmes liés au passage de données entre les vues.

Utopios® Tous droits réservés

Gestion de l'état de l'application :

Utilisation de Combine pour gérer l'état global de l'application

3. Réactivité: En utilisant AppState avec SwiftUI, vous pouvez tirer parti de la réactivité de SwiftUI pour mettre à jour automatiquement les vues lors de la modification de l'état. Cela permet de maintenir la cohérence de l'interface utilisateur et de réduire la complexité de la gestion des mises à jour de l'interface utilisateur.
4. Facilité de test: En regroupant l'état de l'application et la logique métier dans des objets séparés, vous pouvez facilement écrire des tests unitaires pour vérifier le bon fonctionnement de votre code. Cela permet d'assurer la qualité de votre application et de prévenir les régressions.

Utopios® Tous droits réservés

TDD et BDD :

Comment tester les différents éléments des modèles d'architecture

- Unit Testing :nous pouvons tester les éléments individuels de votre architecture en utilisant XCTest, le cadre de test unitaire intégré d'Apple. nous pouvons créer des tests unitaires pour les vues SwiftUI, les modèles de données, et les flux de données Combine. Par exemple, nous pouvons tester si une vue affiche correctement des données fournies par un modèle, ou si un flux de données Combine publie correctement les mises à jour lorsque les données changent.
- UI Testing : En plus des tests unitaires, nous pouvons utiliser XCTest pour créer des tests d'interface utilisateur. Ces tests peuvent simuler des interactions utilisateur avec votre application et vérifier que l'interface utilisateur réagit correctement. Par exemple, nous pouvons tester si l'interaction avec une vue dans SwiftUI déclenche les bonnes actions dans votre application.
- Integration Testing : Les tests d'intégration vérifient que différents composants de notre application fonctionnent correctement ensemble. Par exemple, nous pouvons créer un test d'intégration qui vérifie que les mises à jour de données dans une partie de notre application sont correctement reflétées dans une vue SwiftUI.

Utopios® Tous droits réservés

TDD et BDD :

Comment tester les différents éléments des modèles d'architecture

- Mocking and Stubbing : Lorsque nous testons des parties de notre application, nous voulons souvent isoler la partie que nous testons des autres parties de l'application. Par exemple, si nous testons une vue SwiftUI, nous ne voulons pas dépendre d'un service de données en direct. Pour cela, nous pouvons créer des "mocks" et des "stubs" - des versions simulées de ces services qui retournent des données prédéterminées. Nous pouvons alors utiliser ces mocks et stubs dans nos tests pour nous assurer que notre vue fonctionne correctement indépendamment du service de données.
- Performance Testing : En plus de vérifier la fonctionnalité de notre application, nous voulons souvent aussi vérifier les performances. XCTest comprend des outils pour le test de performance qui vous permettent de mesurer le temps qu'il faut à certaines parties de votre code pour s'exécuter, et de vérifier qu'elles respectent vos objectifs de performance.

Utopios® Tous droits réservés

TDD et BDD :

BDD

BDD est une méthodologie de développement de logiciels qui encourage la collaboration entre développeurs, testeurs de qualité et personnes non techniques ou d'affaires dans un projet de logiciel. Il met l'accent sur l'obtention d'un comportement clair et compréhensible du logiciel.

Utopios® Tous droits réservés

TDD et BDD :

BDD

1. Définir le comportement souhaité: Avec BDD, vous commencez par définir le comportement que vous attendez de votre application. C'est généralement fait en collaboration avec toutes les parties prenantes du projet. Le comportement est défini en termes simples et compréhensibles, souvent en utilisant une syntaxe "Donné que... quand... alors..." (Given-When-Then). Par exemple: "Étant donné que l'utilisateur est sur l'écran d'accueil, quand ils cliquent sur le bouton 'Se connecter', alors ils devraient être dirigés vers l'écran de connexion".
2. Écrire des tests pour ce comportement: Une fois le comportement défini, vous écrivez des tests pour vérifier ce comportement. Swift a un excellent support pour les tests unitaires, et vous pouvez utiliser le XCTest Framework pour écrire vos tests.

Utopios® Tous droits réservés

TDD et BDD :

BDD

3. Développer le comportement: Une fois que vous avez écrit vos tests (et qu'ils échouent, puisque vous n'avez pas encore implémenté le comportement), vous commencez à écrire le code qui fait passer les tests. Avec SwiftUI, vous construisez votre interface utilisateur et, avec Combine, vous gérez les événements asynchrones.
4. Refactoriser: Après avoir écrit le code qui fait passer les tests, vous refactorisez votre code pour le rendre aussi propre et lisible que possible, tout en s'assurant que les tests passent toujours.
5. Répéter: Vous répétez ce processus pour chaque nouveau comportement que vous voulez ajouter à votre application.

Utopios® Tous droits réservés

TDD et BDD :

Exercice

Nous allons créer une application de météo
Voici les comportements que nous voulons :

- L'utilisateur peut entrer le nom d'une ville dans une barre de recherche.
- Après avoir entré le nom de la ville, l'utilisateur voit les informations météorologiques actuelles pour cette ville.
- Les informations météorologiques comprennent la température actuelle, la condition météorologique (par exemple, ensoleillé, nuageux, pluie) et une icône représentant la condition.
- Si l'utilisateur entre le nom d'une ville qui n'existe pas ou qui ne peut pas être trouvée, il voit un message d'erreur.

Utopios® Tous droits réservés