Clean Swift iOS Architecture для виправлення Massive View Controller

Клієнт просить вас оцінити, скільки часу потрібно для виправлення помилки. Ви відповідаєте, що це займе кілька годин, хоча в глибині душі зовсім не впевнені у своїй оцінці. Вам просто потрібно озвучити хоч якусь цифру...

Якщо в реальності на це піде більше часу, ніж ви припустили, клієнт обов'язково зауважить: "Ви казали, що на роботу піде всього кілька годин".

Ви порушили обіцянку і клієнт вам більше не довіряє.

Але найнеприємніше те, що багато розробників схильні недооцінювати час, необхідний для виконання поставлених завдань.

Може проблема не в вас? А клієнти просто не розуміють, як розробляються додатки?

- Вони не знають, наскільки це складно.
- Вони повинні перестати вносити зміни в ТЗ.
- В першу чергу вони повинні зосередитись на функціоналі, а не на дизайні чи інтерфейсі.
- Якби мене попросили зробити це 2 місяці тому, коли я ще добре пам'ятав свій код, його редагування зайняло б набагато менше часу.
- Додайте тут ще кілька своїх улюблених причин.

Ви повинні розібратися у своєму коді. Зорієнтуватися поміж 50 класів, протоколів і методів. Зрозуміти роботу умовних операторів і циклів. Точно визначити рядок, що викликає помилку. Внести зміни в код, щоб знайти її причини. Повторювати ці кроки до тих пір, поки її не виправите.

В процесі виправлень ви постійно будете думати про те, щоб нічого не зламати. Це відбувається тому, що ви не застосовуєте модульне тестування для попередження можливих ускладнень у поведінці додатку.

Ви ходитимете по цьому колу знову і знову, коли потрібно буде виправити якусь помилку або додати новий функціонал. Спасибі **Massive View Controller**.

Невдала спроба вирішення проблеми MVC

Якщо ця ситуація вам знайома, можливо, ви вже намагались вибратися з цього глухого кута. Напевно, ви читали про шаблони проектування та реорганізацію View Controllers. Про те, як застосовувати data sources і delegates. Про винесення бізнес-логіки в моделі. Ці кроки покликані допомогти вам у написанні модульних тестів.

Але ви так і не зробили нічого з перерахованого вище. Вже 2015 рік. Представлені Swift 2.0 і iOS 9.

Речі, про які ви вже встигли щось прочитати, а деякі з них навіть випробувати в своїх проектах, можна умовно назвати терміном “аптечка”. Збиток вже завдано. Ви лікуєте симптоми, а не боретеся з першопричинами.

Чи можемо ми запобігти збитку замість того, щоб спостерігати за його наслідками? Навіщо потрібен цей рефакторинг? Чи можемо ми відразу писати правильний код, який в подальшому не потребуватиме рефакторингу?

Першопричини

Сьогодні до `рефакторингу` і `тестування` прикута увага розробників усього світу. Наскільки реально з їх допомогою писати правильний код, який не потребуватиме `рефакторингу` у майбутньому?

Почнемо з **`архітектури`**. Якщо будівля має слабкий фундамент, вона врешті-решт завалиться. Якщо ваш код має неправильну архітектуру — він буде розростатися. Це означає, що його доведеться переписувати, а це справа не з дешевих.

Існує безліч архітектур, наприклад `MVC`, `MVVM`, `ReactiveCocoa` і `VIPER`, а також різноманітні бібліотеки для тестування. Щоб допомогти вам сьогодні, я хочу зосередитися на тому, як застосовувати ідеї `чистої архітектури` для розробки під `iOS` із використанням мови програмування **`Swift`**. В подальшому скрізь я буду використовувати спеціальний термін **`Clean Swift`**. Після ознайомлення з матеріалами цієї статті ви навчитеся:

- швидко і легко знаходити і виправляти помилки;
- впевнено модифікувати існуючі процеси;
- з легкістю додавати нові можливості;
- писати короткі методи з єдиною відповідальністю;
- в заданих межах розділяти класові залежності;
- переносити усю бізнес-логіку з **`View Controllers`** в **`Interactors`**;

- створювати багаторазові компоненти з об’єктами **worker** і **service**;
- відразу писати правильний код;
- швидко і якісно писати модульні тести, в які можна легко вносити зміни;
- ловити регресії;
- застосовувати отримані знання в нових та існуючих проектах будь-якої складності.

Уявіть собі, що ви точно знаєте, який файл відкрити і який метод переглянути. Наскільки це збільшить продуктивність? Уявіть, що ваші тести запускаються за лічені секунди замість хвилин або й годин очікування. І що вам не потрібно вивчати сторонні бібліотеки для створення модульних тестів або встановлювати CocoaPods.

Для цього потрібно лише дотримуватися єдиної системи і все у вашому коді логічно займе свої місця. При такому підході навіть через три місяці після його написання ви будете точно знати, що і де шукати.

Для того, аби спростити процес входження в дану технологію, існує набір спеціальних Xcode–шаблонів. З їх допомогою ви будете в змозі генерувати нові компоненти архітектури **Clean Swift**, просто обравши в меню пункт New File Command.

У цій статті ми в деталях розглянемо такі компоненти, як: **Interactors**, об’єкти **Workers** і **Service**, подивимось, як працює **Router** з декількома Storyboards.

Вступ у Clean Swift Architecture for iOS

Архітектура **Clean Swift** є похідною від Clean Architecture, яку свого часу запропонував Uncle Bob. Вони мають багато спільних понять, таких як компоненти, межі і моделі. Я покажу вам, як застосовувати Clean Architecture в iOS-проектах із використанням **Swift**.

Кейс “Створити замовлення”

Дані:

- ідентифікатор клієнта;
- контактна інформація про клієнта;
- адреса доставки;
- спосіб доставки;
- спосіб оплати.

Основний процес:

1. Оператор виконує команду “Створити замовлення” із зазначеними вище даними.
2. Система перевіряє введені дані.
3. Система створює замовлення і генерує для нього персональний ідентифікатор.
4. Система повертає оператору присвоєний замовленню ідентифікатор.

Процес обробки помилок:

1. Система повертає оператору повідомлення про помилку.

У цій моделі ми будемо моделювати дані для обробки Замовлення, а також створимо спеціальні об'єкти типу: запит (request), відповідь (responce) і модель представлення (view model) для передачі даних поміж компонентами **View Controller**, **Interactor** і **Presenter**.

Кожен пункт описаної вище бізнес-логіки ми реалізуємо за допомогою компонента **Interactor** і, в разі необхідності, створимо декілька **Workers**.

Для початку, давайте розглянемо, як краще організувати наш код в середовищі Xcode.

Організація коду в Xcode

Настав час запустити Xcode і створити новий проект під назвою CleanStore. Оберіть Single View Application і iPhone у якості підтримуваної платформи. Переконайтеся, що встановлена мова **Swift**. Створіть основну папку Scenes і вкладену підпапку CreateOrder. У майбутньому для реалізації кейса "Видалення замовлення" ми створимо в нашій основній групі папку з ім'ям DeleteOrder.

Серед програмістів заведено групувати файли проекту згідно парадигми MVC. Досить часто в Xcode Project Navigation можна побачити папки з такими назвами, як Model, View і Controller. Таке іменування нічого не скаже про сам проект. Згідно з постулатами від Uncle Bob назви груп та файлів повинні розкривати наміри, вказувати на їх призначення. Вони не повинні повторювати структуру використовуваних бібліотек або патернів програмування. Тому всередині Scenes ми будемо створювати нові папки, що будуть описувати завдання, які вони вирішують.

Наприклад, всередині групи CreateOrder будуть розміщені всі файли, що мають відношення до кейса “Створення замовлення”. Швидше за все, в групі DeleteOrder ви знайдете код, який буде забезпечувати кейс “Видалення замовлення”. Беручи до уваги таку логіку можна сміливо припустити, які саме файли будуть розміщені у папці ViewOrderHistory, створеній іншим програмістом.

Така організація файлів в проекті дасть вам набагато більше, ніж загальноприйняте групування Model–View–Controller. З часом у вас може з’явитися 15 папок Model, 27 папок View Controllers і 17 папок Views. Як зрозуміти, що відбувається в кожній із них? Щоб відповісти на ці питання, доведеться їх всі переверіти!

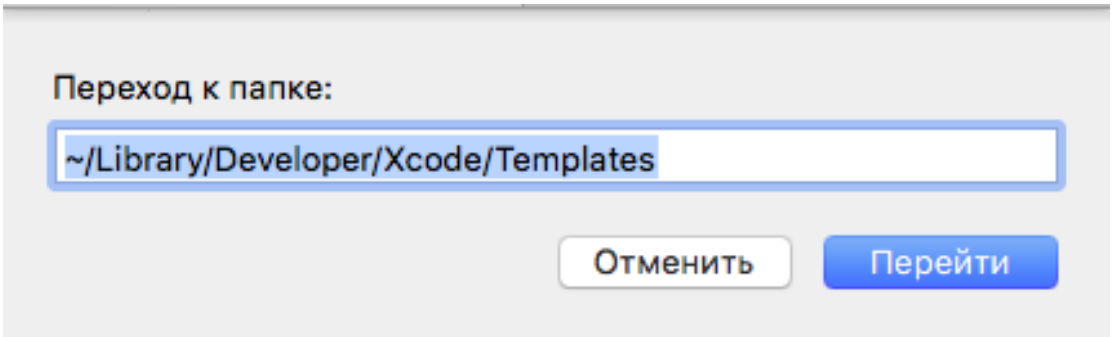
А як щодо використання спільних класів і протоколів для кейсів CreateOrder, DeleteOrder і ViewOrderHistory? Як варіант, ви можете помістити їх в окрему групу Order всередині папки Common ...

Але повернімося до нашого кейса.

Спочатку всередині групи CreateOrder створимо основні компоненти архітектури **Clean Swift**. Потім у вхідні і вихідні протоколи додамо методи, які реалізують нашу бізнес-логіку.

Але як саме за допомогою готових шаблонів створити сім чистих компонентів **Clean Swift**?

Щоб додати новий Template в середовище розробки Xcode необхідно відкрити програму Finder і через меню “Перехід / Перехід до папки” вивести на екран форму для введення шляху до папки, яка містить всі Templates:

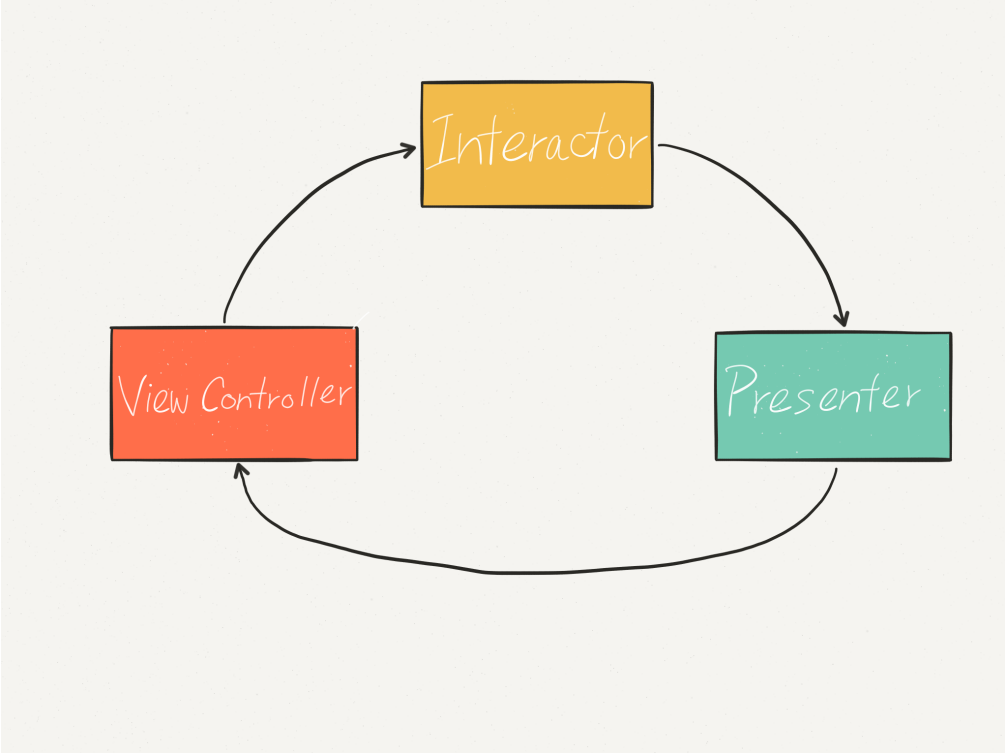


Сюди ми скопіюємо нашу папку з шаблонами.

Тепер ми можемо скомпілювати проект і розпочати створення інтерфейсу користувача за допомогою `Storyboard`. Але для початку давайте розглянемо створені нами компоненти **`Clean Swift`**.

VIP-цикл

`View Controller-Interactor-Presenter` — це три основні компоненти архітектури **`Clean Swift`**.



На малюнку зображено, як вони пов’язані між собою.

Вихід **View Controller** підключається до входу компонента **Interactor**.
Вихід **Interactor** підключається до входу **Presenter**.
Вихід **Presenter** підключається до входу **View Controller**.

Ми створимо спеціальні об’єкти для передачі даних за межі названих компонентів. Це дозволить нам відокремити їх базові моделі даних. Ці спеціальні базові моделі повинні складатися тільки з примітивних (простих) типів, таких як Int, Double і String. Для представлення даних ми можемо створити структури, класи або перерахування, але необхідною умовою є використання тільки примітивних типів всередині цих трьох сутностей.

Це важливо, оскільки будь-які зміни логіки бізнес-процесів неминуче призведуть до змін в базових моделях даних. При такому підході нам не доведеться оновлювати весь код. Компоненти **Clean Swift** виступають у якості плагінів. Це означає, що ми можемо змінювати їх за умови, що не порушуються встановлені відповідності між вхідними та вихідними протоколами. Тільки в такому випадку наш додаток продовжить працювати, як і було задумано при його створенні.

Стандартний сценарій виглядає так. Користувач натискає кнопку на екрані. Оброблювач жестів через IBActions спрацьовує у **View Controller**, який створює об’єкт–запит (request) і передає його на вхід **Interactor**. Компонент **Interactor** приймає об’єкт–запит (request) і виконує певну роботу. По її закінченню він розміщує результат своїх обчислень в об’єкт–відповідь (responce) і передає його на вхід **Presenter**. Останній приймає об’єкт–відповідь (responce) і формує (форматує) кінцеві результати.

Наприкінці створюється об'єкт–модель представлення даних (view model), яка згідно **VIP–циклу** повертається назад у **View Controller**. І, нарешті, користувачу показується результат виконання.

1. View Controller

Що в додатку iOS повинен робити **View Controller**? Відповісти на це питання допоможе базовий клас UITableViewController. Ви хочете передати управління кодом в підкласи UITableView і UIView? Але як це виглядає на практиці? І що можна вважати "кодом управління", а що ні?

Давайте розбиратися.

```
import UIKit

protocol CreateOrderViewControllerInput {
     func displaySomething(viewModel: CreateOrderViewModel)
}

protocol CreateOrderViewControllerOutput {
     func doSomething(request: CreateOrderRequest)
}

class CreateOrderViewController: UITableViewController,
CreateOrderViewControllerInput {
     var output: CreateOrderViewControllerOutput!
     var router: CreateOrderRouter!
```

```
    // MARK: Object lifecycle
    override func awakeFromNib()  {
        super.awakeFromNib()

CreateOrderConfigurator.sharedInstance.configure(self)
    }

    // MARK: View lifecycle
    override func viewDidLoad()  {
        super.viewDidLoad()
        doSomethingOnLoad()
    }

    // MARK: Event handling
    func doSomethingOnLoad()  {
        // NOTE: Ask the Interactor to do some work
        let request = CreateOrderRequest()
        output.doSomething(request)
    }

    // MARK: Display logic
    func displaySomething(viewModel: CreateOrderViewModel)
{
        // NOTE: Display the result from the Presenter

        // nameTextField.text = viewModel.name
    }
}
```

Протоколи CreateOrderViewControllerInput та CreateOrderViewControllerOutput

Протокол CreateOrderViewControllerInput описує, як компонент **CreateOrderViewController**, який повинен відповідати даному протоколу, буде працювати зі своїми входами. Протокол CreateOrderViewControllerOutput описує, як цей же компонент буде працювати зі своїми виходами. Пізніше ви зможете побачити аналогічні протоколи у компонентів **Interactor** і **Presenter**.

Вихідний протокол містить метод doSomething(). Якщо інший компонент хоче виступити у якості виходу для **CreateOrderViewController**, він повинен підтримувати doSomething() на своєму вході.

Згідно логіки **VIP-циклу** вихід **View Controller** взаємодіє з входом **Interactor**. Зверніть увагу, що у файлі CreateOrderViewController.swift немає згадки імені компонента **CreateOrderInteractor**. Це означає, що ми можемо наказати іншому компоненту стати входом для **CreateOrderViewController** за умови підтримки методу doSomething() у своєму вхідному протоколі.

У методі doSomething() аргументом виступає об'єкт-запит (request), який передається з **View Controller** в **Interactor**. Цей об'єкт-запит (request) вдає із себе структуру CreateOrderRequest. Вона складається з примітивних типів і не включає в себе всі дані Замовлення. Це означає, що ми повинні відокремити моделі даних **View Controller** і **Interactor**. Пізніше, коли ми вирішимо внести зміни в існуючу модель даних (наприклад, додати нове поле з внутрішнім ID замовлення), нам не доведеться нічого міняти в компонентах **Clean Swift**.

Після закінчення розгляду **VIP-циклу** ми повернемося до методу displaySomething() нашого вихідного протоколу.

Змінні output і router

Змінна output вдає із себе об'єкт, який відповідає протоколу CreateOrderViewControllerOutput. Хоча ми знаємо, що на прийомі повинен бути **Interactor**, це не завжди так.

Змінна router посилається на **CreateOrderRouter** і використовується для організації переходів між різними сценами.

Метод configure()

Ми викликаємо CreateOrderConfigurator.sharedInstance.configure(self) в методі awakeFromNib(), щоб вказати **Configurator**'у **VIP-ціль**. Сам компонент **Configurator** ми розглянемо пізніше.

Потік управління

У розділі viewDidLoad() ми викликаємо метод нашої бізнес-логіки під назвою doSomethingOnLoad(). Цей метод створює об'єкт CreateOrderRequest() і викликає doSomething(request) на вході **Interactor**. От і все. На виході ми відпрацьовуємо нашу бізнес-логіку. **View Controller** не хвилює, хто і як це зробив.

2. Interactor

Interactor містить в собі бізнес-логіку програми. Взаємодія з додатком відбувається через різні види торкання екрана у поєднанні з жестами. **View Controller** збирає дані про такі події і передає їх в **Interactor**, який отримує модель даних і передає її для обробки в різні обробники (**workers**).

```
import UIKit

protocol CreateOrderInteractorInput {
     func doSomething(request: CreateOrderRequest)
}

protocol CreateOrderInteractorOutput {
     func presentSomething(response: CreateOrderResponse)
}

class CreateOrderInteractor: CreateOrderInteractorInput {
     var output: CreateOrderInteractorOutput!
     var worker: CreateOrderWorker!

     // MARK: Business logic
     func doSomething(request: CreateOrderRequest)  {
          // NOTE: Create some Worker to do the work
          worker = CreateOrderWorker()
          worker.doSomeWork()

          // NOTE: Pass the result to the Presenter
          let response = CreateOrderResponse()
          output.presentSomething(response) } }
```

Протоколи CreateOrderInteractorInput і CreateOrderInteractorOutput

Протокол CreateOrderInteractorInput описує, як буде працювати з входами компонент **CreateOrderInteractor**, який повинен йому відповідати. Протокол CreateOrderInteractorOutput описує, як цей же компонент буде працювати з виходами.

Аналогічно протоколу CreateOrderViewControllerOutput, всередині чергового протоколу CreateOrderInteractorInput ми бачимо вже знайомий нам метод doSomething(). Вихід **CreateOrderViewController** підключений до входу **CreateOrderInteractor**.

Вихідний протокол містить в собі один метод presentSomething(). Вихід компонента **CreateOrderInteractor** повинен підтримувати presentSomething() для того, щоб діяти у якості вихідного. Підказка: вихід перетворюється у **P** згідно **VIP**.

Також варто звернути увагу на аргумент в методі doSomething() — це об'єкт-запит типу CreateOrderRequest, який ми описали в протоколі CreateOrderViewControllerOutput. Для виконання своєї роботи компонент **Interactor** бере необхідні йому дані з цього об'єкта-запита (request).

Так само для методу presentSomething() аргументом виступає той самий об'єкт-відповідь (response) типу CreateOrderResponse.

Змінні output і worker

Змінна `output` вдає із себе об'єкт, який відповідає протоколу `CreateOrderViewControllerOutput`. Хоча ми знаємо, що на прийомі повинен бути **Presenter**, це не завжди так.

Змінна `worker` типу **CreateOrderWorker** — це спеціальний об'єкт, який реально буде створювати нове Замовлення. Кейс "Створення замовлення" передбачає збереження даних в базі **Core Data** і виконання мережевих запитів. **Interactor** складно самостійно впоратися з такою кількістю завдань. Крім того, **Interactor** повинен ще перевірити форму Замовлення. Швидше за все, виконанням цієї роботи займеться окремий `worker`.

Потік управління

На вході компонента **CreateOrderInteractor** викликається метод `doSomething()`, який спочатку створює об'єкт **Worker**, щоб потім передати йому на виконання роботи, описані в іншому методі `doSomeWork()`. На завершення створюється об'єкт–відповідь (response) і на виході викликається метод `presentSomething()`.

Давайте розглянемо об'єкт **Worker**.

3. Worker

На екрані профілю користувача необхідно витягти його персональні дані з бази **Core Data**, скачати фото, дозволити користувачеві ставити like, підписатися на отримання новин і т.д. Ми не будемо засмічувати **Interactor** таким функціоналом. Тому виконання кожного завдання доручається окремому **Worker**. У подальшому ми зможемо повторно використовувати вже створені **Workers** в інших місцях нашого проекту.

Використовувати компонент **CreateOrderWorker** для виконання завдань в **Interactor** дуже зручно, оскільки він просто надає інтерфейс і реалізацію своєї роботи.

```
import UIKit

class CreateOrderWorker {
     // MARK: Business Logic
     func doSomeWork()  {
          // NOTE: Do the work
     }
}
```

4. Presenter

Після того, як **Interactor** закінчить обчислення, він передасть результати роботи у вигляді об'єкта-відповіді (response) на вхід компонента **Presenter**. Залежно від того, з якого екрана була викликана на обробку відповідна бізнес-логіка **Presenter** перетворить об'єкт-відповідь (response) в об'єкт-модель представлення даних (view model).

На виході готова модель повертається на вхід **View Controller**, щоб показати ці результати користувачу.

```
import UIKit

protocol CreateOrderPresenterInput {
     func presentSomething(response: CreateOrderResponse)
}

protocol CreateOrderPresenterOutput: class {
     func displaySomething(viewModel: CreateOrderViewModel)
}

class CreateOrderPresenter: CreateOrderPresenterInput {
     weak var output: CreateOrderPresenterOutput!

     // MARK: Presentation logic
     func presentSomething(response: CreateOrderResponse)  {
          // NOTE: Format the response from the Interactor and
pass the result back to the View Controller
          let viewModel = CreateOrderViewModel()
          output.displaySomething(viewModel)
     }
}
```

Протоколи CreateOrderPresenterInput і CreateOrderPresenterOutput

Протокол CreateOrderPresenterInput визначає входи для компонента **CreateOrderPresenter**, який повинен йому відповідати. Протокол CreateOrderPresenterOutput визначає виходи для цього ж компонента.

Думаю, що вже відомі нам методи presentSomething() і displaySomething() більше не потребують якихось додаткових пояснень. Аргумент CreateOrderResponse передається через межі компонентів **Interactor–Presenter**, в той час як аргумент CreateOrderViewModel передається через межі **Presenter–View Controller** в момент завершення **VIP–циклу**.

Змінна output

Змінна output вдає із себе об'єкт, який відповідає протоколу CreateOrderViewControllerOutput. Хоча ми знаємо, що на прийомі повинен бути **View Controller**, це не завжди так. Невелика відмінність: в даному випадку ми робимо слабке посилання на змінну output, щоб уникнути утримання сцени CreateOrder в момент, коли вона більше не потрібна і компоненти повинні бути вивільнені.

Потік управління

Так як вихід компонента **CreateOrderInteractor** підключений до входу компонента **CreateOrderPresenter**, метод presentSomething() буде викликаний після того, як **Interactor** закінчить свою роботу. Саме тоді створюється об'єкт–модель представлення даних (view model) і викликається вихідний метод displaySomething().

Настав час описати метод displaySomething() для **View Controller**. Це останній етап **VIP–циклу**. Він приймає будь-які об'єкти у вигляді моделі представлення даних (view model) і показує їх користувачеві. Наприклад, в текстовому полі нам потрібно показати ім'я користувача:

```
nameTextField.text = viewModel.name.
```

Вітаємо! Тепер ви розумієте суть **Clean Swift**. Ви в змозі розібратися у бізнес-лозіці, закладеній у програмному коді. Але давайте спочатку закінчимо розгляд інших компонентів, якіс стосуються архітектури **Clean Swift**.

5. Router

Коли користувач натискає кнопку Next, щоб перейти до наступного екрана (сцени) зі Storyboard, спрацьовує Segue і відображається **View Controller**. **Router** витягує всю необхідну навігаційну логіку з **View Controller**. Такий компонент — найкращий спосіб для передачі будь-яких даних поміж екранами (сценами). В результаті перед **View Controller** стоїть лише одне завдання — контроль представлень (views).

```
import UIKit

protocol CreateOrderRouterInput {
    func navigateToSomewhere()
}

class CreateOrderRouter {
    weak var viewController: CreateOrderViewController!

    // MARK: Navigation
    func navigateToSomewhere()  {
        // NOTE: Teach the router how to navigate to another scene. Some examples follow:

        // 1. Trigger a storyboard segue
        // viewController.performSegueWithIdentifier("ShowSomewhereScene", sender: nil)

        // 2. Present another view controller programmatically
        // viewController.presentViewController(someWhereViewController, animated: true, completion: nil)

        // 3. Ask the navigation controller to push another view controller onto the stack
        // viewController.navigationController?.pushViewController(someWhereViewController, animated: true)
```

```
        // 4. Present a view controller from a different
storyboard
        // let storyboard = UIStoryboard(name:
"OtherThanMain", bundle: nil)
        // let someWhereViewController =
storyboard.instantiateInitialViewController() as!
SomeWhereViewController
        //
viewController.navigationController?.pushViewController(someW
hereViewController, animated: true)
    }

    // MARK: Communication
    func passDataToNextScene(segue: UIStoryboardSegue)  {
        // NOTE: Teach the router which scenes it can
communicate with
        if segue.identifier == "ShowSomewhereScene" {
                passDataToSomewhereScene(segue)
    }
  }

  func passDataToSomewhereScene(segue: UIStoryboardSegue) {
    // NOTE: Teach the router how to pass data to the next
scene
    // let someWhereViewController =
segue.destinationViewController as! SomeWhereViewController
    // someWhereViewController.output.name =
viewController.output.name
  }
}
```

Протокол CreateOrderRouterInput

Протокол `CreateOrderRouterInput` описує наші маршрути — переходи на екрани (сцени) або на **View Controller**. Компонент **View Controller** використовує метод `navigateToSomewhere()` у якості маршрутизатора, який знає, як переміщатися поміж різними екранами (сценами).

У коментарі всередині методу `navigateToSomewhere()` зазначено, що компонент **Router** показує гнучкість стосовно можливих варіантів переходу на інші екрани (сцени). Більш детально про **Router** я розповім в іншій статті.

Змінна viewController

Змінна `viewController` — це просте посилання на **View Controller**, який використовує даний **Router**. Щоб уникнути появи циклічних посилань ця змінна є слабкою і встановлюється в конфігураторі. Для переходів між екранами (сценами) компанія **Apple** застосовує спосіб, при якому в клас `UIViewController` додаються всі `present *` і `push *` методи. Тому ми повинні мати під рукою змінну `viewController`, щоб викликати ці методи в **Router**.

passDataToNextScene() і passDataToSomewhereScene()

Методи `passDataToNextScene()` і `passDataToSomewhereScene()` дозволяють передавати дані на наступний екран (сцену). Тут можна використовувати код, який ми поміщаємо всередину методу `prepareForSegue()`.

passDataToNextScene() зіставляє ідентифікатор переходу із заданим умовою значенням, щоб викликати більш специфічний метод passDataToSomewhereScene(), який займається передачею данних.

6. Configurator

Завдання **Configurator** полягає в тому, щоб налаштувати всі описані вище компоненти **Clean Swift**. Всі налаштування можна витягнути з **Configurator**, тому ви можете зосередитися на написанні коду програми. Малоймовірно, що вам коли-небудь доведеться переглядати в **Configurator** всі компоненти. Але спробую трохи підняти завісу таємності :)

```
import UIKit

// MARK: Connect View, Interactor, and Presenter
extension CreateOrderViewController:
CreateOrderPresenterOutput {
    override func prepareForSegue(segue: UIStoryboardSegue,
sender: AnyObject?)  {
        router.passDataToNextScene(segue)
    }
}

extension CreateOrderInteractor:
CreateOrderViewControllerOutput {
}
```

```
extension CreateOrderPresenter: CreateOrderInteractorOutput {
}

class CreateOrderConfigurator {
     // MARK: Object lifecycle
     class var sharedInstance: CreateOrderConfigurator  {
          struct Static {
                    static var instance:
CreateOrderConfigurator?
                    static var token: dispatch_once_t = 0
          }

          dispatch_once(&Static.token) {
                    Static.instance = CreateOrderConfigurator()
          }

          return Static.instance! }
     // MARK: Configuration
     func configure(viewController:
CreateOrderViewController)  {
          let router = CreateOrderRouter()
          router.viewController = viewController

          let presenter = CreateOrderPresenter()
          presenter.output = viewController

          let interactor = CreateOrderInteractor()
          interactor.output = presenter

          viewController.output = interactor
          viewController.router = router } }
```

Розширення

Пам'ятаєте, я писав, що у **View Controller** немає ніяких згадок про **Interactor**. Змінна output має тип CreateOrderViewControllerOutput. На виході **View Controller** ви можете замінити **Interactor** будь-яким іншим об'єктом. Одначе **View Controller** повинен бути підключений до **Interactor**-а, вірно? За цим стежать три наших розширення.

Розширення **CreateOrderViewController** додає протокол відповідності CreateOrderPresenterOutput. Це означає, що вихід **Presenter** підключений до **View Controller**.

Аналогічно, розширення **CreateOrderInteractor** додає протокол відповідності CreateOrderViewControllerOutput.

Розширення **CreateOrderPresenter** додає протокол відповідності CreateOrderInteractorOutput.

Singleton

У проекті має бути тільки один **конфігуратор** для кожного екрана (сцени) і код настройки з'єднання повинен виконатися тільки один раз. Тому для **View Controller** я створюю змінну класу sharedInstance і всередині методу awakeFromNib() викликаю метод ініціалізації:

```
CreateOrderConfigurator.sharedInstance.configure(self)
```

Це найперше, що необхідно зробити відразу після запуску **View Controller** зі Storyboard. Також можна перевизначити базовий клас UIViewController і створювати цю змінну в ньому, але не будемо цього робити.

Метод configure()

Давайте подивимось на стрілки у **VIP–циклі**. Вони односпрямовані. Такий послідовний потік управління робить схему більш зрозумілою. Ось чому ви точно знаєте, який файл і метод потрібно правити для пошуку та виправлення помилок.

Тільки **View Controller** завантажується зі Storyboard. Нам залишається вручну створити екземпляри **Interactor**, **Presenter** і **Router**. За допомогою методу configure() ми створюємо їх і потім присвоюємо відповідні посилання на змінні **output**, r**outer** і **viewController**.

Головне — пам'ятати принцип роботи **VIP–циклу**:

Вихід **View Controller** з'єднаний з входом **Interactor**.
Вихід **Interactor** з'єднаний з входом **Presenter**.
Вихід **Presenter** з'єднаний з входом **View Controller**.

Це означає, що потік управління завжди спрямований в один бік.

VIP–цикл стане у нагоді під час реалізації функцій та виправлення помилок. Ви будете точно знати, який файл і метод слід шукати.

Він також спрощує наш граф залежностей. Ви ж не хочете, щоб об'єкти посилались одне на одного, коли їм заманеться. Вам може здаватися, що це зручно, коли **View Controller** безпосередньо вказує **Presenter**-у на необхідність відформатувати рядок. Але вкінці-кінців такий підхід спричинить хаос всередині вашого графа залежностей. Пам'ятайте про це, щоб уникнути створення непотрібних зв'язків.

Це застереження стане більш зрозумілим, коли ми реалізуємо наш перший кейс під назвою "Створення Замовлення".

7. Моделі

Для того, щоб повністю відокремити усі компоненти **Clean Swift** один від одного, ми повинні визначити моделі даних, які будуть передаватися через межі між ними, а не просто використовувати "сирі" моделі даних.

Існують 3 основних типи моделей:

Запит (request) — **View Controller** створює модель-запит (request model) і передає її на вхід **Interactor**. Як правило, модель-запит (request model) містить введені користувачем дані, такі як: значення текстових полів + дані, обрані за допомогою елемента управління UIPickerView.

Відповідь (response) — після того, як **Interactor** закінчить обробку моделі-запиту (request model), він інкапсулює результати у модель-відповідь (response model) і передасть її на вхід **Presenter**.

Модель-представлення (view model) — після того, як **Presenter** отримав модель-відповідь (response model) від **Interactor**, він приступає до форматування результатів в примітивні типи даних, такі як: `String` і `Int`, а потім формує з них модель-представлення (view model). Далі він відсилає свою модель назад на вхід **View Controller**, щоб останній показав її користувачу.

```
struct CreateOrderRequest {
}

struct CreateOrderResponse {
}

struct CreateOrderViewModel {
}
```

У коді, створеному на базі шаблонів, у всіх моделях відсутні будь-які фактичні дані. Це просто порожні моделі. Вони наповняться актуальними даними після того, як ми реалізуємо бізнес-логіку кейса “Створення Замовлення”. Давайте подивимося, як виглядають ці дані.

Дані для кейса “Створення Замовлення”

Нижче перераховано початкові дані, необхідні для створення нового Замовлення. Використовуючи таблицю і текстові поля за допомогою Storyboard ми створимо форму, в яку користувач буде вводити значення таких даних:

- Customer ID
    - Integer
- Customer Contact Info
    - First name
    - Last name
    - Phone
    - Email
- Shipment Destination
    - Shipping address
        - Street 1
        - Street 2
        - City
        - State
        - ZIP
- Shipment Mechanism
    - Shipping method
        - Next day
        - 3 to 5 days
        - Ground
- Payment Information
    - Credit card number
    - Expiration date
    - CVV
    - Billing address
        - Street 1
        - Street 2
        - City
        - State
        - ZIP

Це буде гігантський екран! Але у реальному додатку частину цих даних можна отримати одразу після того, як користувач увійде в систему: `Name`, `Phone`, `Email`, `Shipping`, `Billing address` і можливо `Credit card info`. Тому кінцева форма вийде значно меншою.

Бізнес-логіка кейса “Створення Замовлення”

Тепер давайте подивимося, яку бізнес-логіку ми можемо вигадати відповідно до наших вимог. Вона може слугувати псевдокодом для створення приймальних тестів у майбутньому:

Сформувати форму кейса “Створення Замовлення” з наведеними вище даними.

- Показати форму для збору даних від користувача.
- Форма використовує текстові поля для збору текстових даних.
- Форма використовує елемент `UIPickerView` для введення даних у поля `Shipping method` і `Expiration`.
- Форма використовує елемент `UISwitch` для автоматичного заповнення поля `Billing address` значенням з поля `Shipping address`.
- Форма використовує елемент `UIButton` для запуску виконання кейса “Створення Замовлення”.

Виконати валідацію всіх даних.

- Переконатися, що всі поля, крім `Street 2` непорожні.
- Відобразити повідомлення “Валідний” поруч з елементом `UIButton`.
- Відобразити повідомлення про помилку валідації поруч з відповідними полями.

Згенерувати для нового Замовлення унікальний ID-номер.

- Створити нове Замовлення і привласнити йому ID-ідентифікатор.
- Створити і зберегти нове Замовлення в **Core Data**.
- Доставити ідентифікатор Замовлення оператору.
- Відобразити на екрані номер нового Замовлення для користувача.

Це хороший набір первинних функцій, щоб продемонструвати роботу **Clean Swift** та його переваги. У майбутніх статтях я розширю цей набір, щоб показати, як легко можна адаптуватися до змін. Ось деякі з них:

- Використання Core Location для зворотного геокодування поточних координат в адресу з метою попереднього заповнення даних у полях Shipping address та Billing address.
- Інтеграція Stripe API для збору інформації про кредитні картки.
- Перевірка введенного значення безпосередньо під час вводу замість того, аби чекати, коли буде натиснута кнопка.
- Додавання назви країни в адреси Shipping і Billing з метою розширення бізнесу клієнта за кордоном.
- Форматування даних у полях Phone number, Credit card number і Expiration date.

Зараз ми готові реалізувати кейс “Створення Замовлення”.

Дизайн форми “Створення Замовлення” в Storyboard і View Controller

З чого почнемо?

Спочатку користувач вводить початкові дані у компоненті **View Controller**. Далі **контролер** передає зібрану інформацію на вхід **Interactor**. З виходу **Interactor**-а дані передаються на вхід **Presenter** для форматування. І нарешті, **Presenter** пропонує **View Controller** відобразити отримані результати.

Ми ще раз описали потік управління, який завжди виконується в одному напрямку. Назва **VIP–цикл** походить від перших літер у назві компонентів, які його формують, а також вказує на порядок їх взаємодії між собою:

V –> I –> P.

Давайте перейдемо до створення екрана (сцени) з кейса “Створення Замовлення”. Для цього в Storyboard розмістимо контролер **CreateOrderViewController**, “загорнутий” у UINavigationController. Присвоїмо заголовку нашої форми назву “Створити Замовлення”.

Створимо табличне представлення за допомогою статичних рядків. Згрупуємо дані в секції (розділи), де кожному значенню з кейса буде відповідати окремий рядок. Додамо в рядок елементи UILabel і UITextField.

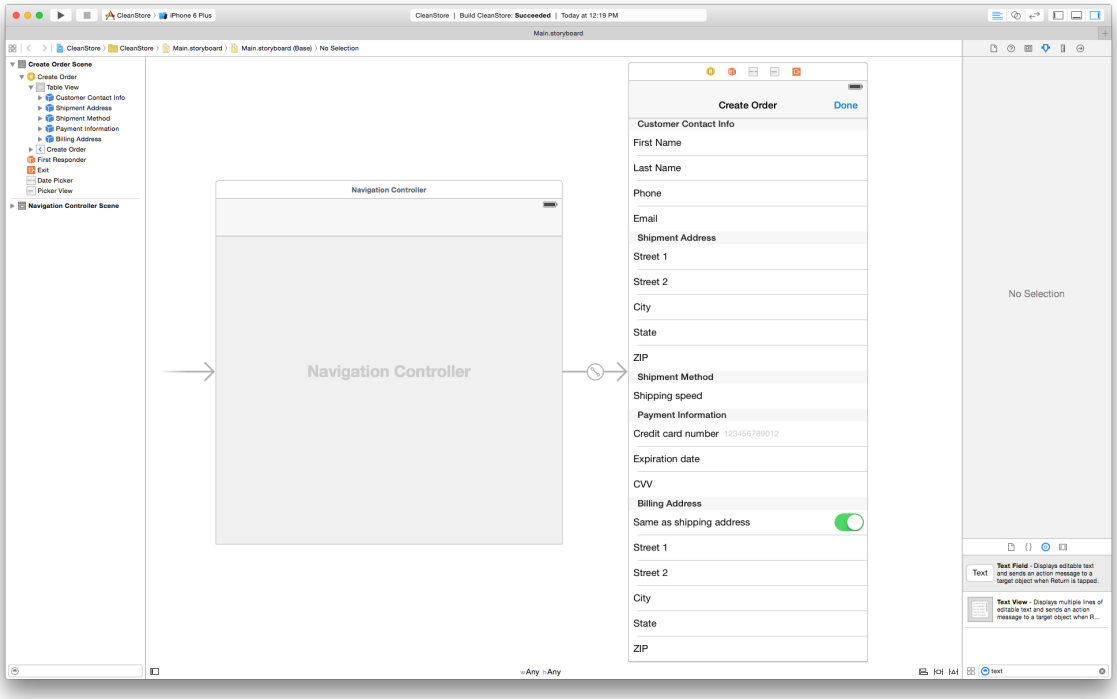
Для текстових полів створимо відповідні IBOutlet-з'єднання:

- textFields — для всіх текстових полів. Встановимо, що **CreateOrderViewController** виступає їхнім делегатом.
- shippingMethodTextField — для текстового поля Shipping method.
- expirationDateTextField — для текстового поля Expiration date.

Із Object Library додамо на екран (сцену) елементи управління UIPickerView і UIDatePicker, для яких створимо відповідні IBOutlet-з'єднання:

- shippingMethodPicker = елемент управління UIPickerView. Зазначимо, що контролер **CreateOrderViewController** виступає для нього одночасно і datasource, і delegate.
- expirationDatePicker = елемент управління UIDatePicker. Створимо також обробник подій IBAction, пов'язаний з даним елементом і назвемо його expirationDatePickerValueChanged().

Наш екран (сцена) повинен виглядати наступним чином:



Можливо він не тягне на премію за найкращий дизайн, проте його функціонал повністю задовольняє вимоги нашого кейса. Краса **Clean Swift** полягає у тому, що ви можете модифікувати цей екран (сцену) пізніше, не чіпаючи інші частини програми. Наприклад, ми можемо додати назву країни в Shipping і Billing address для розширення клієнтського бізнесу за кордоном. Або ми можемо найняти професійного дизайнера для створення нового стилю нашої форми.

Після настройки IBOutlets і IBActions, контролер **CreateOrderViewController** повинен містити в собі наступний код:

```
// MARK: Text fields
@IBOutlet var textFields: [UITextField]!
// MARK: Shipping method
@IBOutlet weak var shippingMethodTextField: UITextField!
@IBOutlet var shippingMethodPicker: UIPickerView!
```

```
// MARK: Expiration date
@IBOutlet weak var expirationDateTextField: UITextField!
@IBOutlet var expirationDatePicker: UIDatePicker!

@IBAction func expirationDatePickerValueChanged(sender:
AnyObject)  {
}
```

Тепер інтерфейс користувача готовий. Давайте налаштуємо способи його взаємодії з екраном (сценою). Нам потрібно задати переходи між полями для введення даних при натисканні користувачем кнопки `Next` на екранній клавіатурі. Для цього зазначимо, що **`CreateOrderViewController`** повинен відповідати протоколу `UITextFieldDelegate`. Також потрібно додати метод `textFieldShouldReturn()`.

```
func textFieldShouldReturn(textField: UITextField) -> Bool  {
     textField.resignFirstResponder()

     if let index = textFields.indexOf(textField) {
               if index < textFields.count - 1 {
                    let nextTextField = textFields[index + 1]
                    nextTextField.becomeFirstResponder()
               }
     }

     return true
}
```

Коли користувач натискає на табличний рядок, необхідно зробити так, щоб текстове поле у вибраному рядку переходило в режим редагування. Встановимо для `UITextField` властивість `UITextBorderStyleNone`, *а* також додамо метод `tableview:didSelectRowAtIndexPath()`:

```
override func tableView(tableView: UITableView,
didSelectRowAtIndexPath indexPath: NSIndexPath)   {
     if let cell = tableView.cellForRowAtIndexPath(indexPath)
{
          for textField in textFields {
               if textField.isDescendantOfView(cell) {
                    textField.becomeFirstResponder()
               }
          }
     }
}
```

Реалізація бізнес-логіки “Методи доставки" у Interactor

У полях `shipping method` і `expiration date` для вибору відповідного значення задіяні `пікери`. Саме тут ми вперше зустрічаємося з елементами нашої бізнес-логіки.

Давайте спочатку розглянемо метод `shipping`.

З плином часу способи доставки можуть змінюватися, тому ми винесемо цю бізнес-логіку з **`View Controller`** і перенесемо її в **`Interactor`**.

Для цього виконаємо метод configurePickers() з розділу viewDidLoad(). З точки зору інтерфейсу користувача у момент взаємодії з полем shippingMethodTextField правильним буде використати елемент управління UIPickerView замість стандартної клавіатури:

```
override func viewDidLoad() {
     super.viewDidLoad()
     configurePickers()
}

func configurePickers() {
     shippingMethodTextField.inputView = shippingMethodPicker
}
```

Далі встановимо для CreateOrderViewController відповідність протоколам UIPickerViewDataSource і UIPickerViewDelegate та додамо наступні методи:

```
func numberOfComponentsInPickerView(pickerView: UIPickerView)
-> Int {
     return 1
}

func pickerView(pickerView: UIPickerView,
numberOfRowsInComponent component: Int) -> Int {
     return output.shippingMethods.count
}
```

```
func pickerView(pickerView: UIPickerView, titleForRow row:
Int, forComponent component: Int) -> String? {
    return output.shippingMethods[row]
}

func pickerView(pickerView: UIPickerView, didSelectRow row:
Int, inComponent component: Int) {
    shippingMethodTextField.text =
output.shippingMethods[row]
}
```

Оскільки ми перенесли обробку бізнес-логіки процесу доставки в **Interactor**, компонент **View Controller** повинен використовувати метод `output.shippingMethods` для її виклику. Крім того, нам потрібно додати змінну `shippingMethods` в протоколи `CreateOrderViewControllerOutput` і `CreateOrderInteractorInput`. Ми повинні додати її в обидва протоколи, оскільки **Configurator** з'єднує вихід **CreateOrderViewController** з входом **CreateOrderInteractor**. На даний момент ми маємо в своєму розпорядженні масив строкових літералів. У майбутньому ми зможемо динамічно отримувати можливі способи доставки з **Core Data** або через мережу.

CreateOrderViewController:

```
protocol CreateOrderViewControllerOutput {
    var shippingMethods: [String] { get }
}
```

CreateOrderInteractor:

```
protocol CreateOrderInteractorInput {
     var shippingMethods: [String] { get }
}

class CreateOrderInteractor: CreateOrderInteractorInput {
     var output: CreateOrderInteractorOutput!
     var worker: CreateOrderWorker!
     var shippingMethods = [
          "Standard Shipping",
          "Two-Day Shipping ",
          "One-Day Shipping "
     ]
}
```

Реалізація бізнес-логіки Expiration Date в Interactor

Значення поля `Expiration date` (дата закінчення замовлення) може бути відформатовано по-різному в залежності від мовних налаштувань девайса користувача та його місцезнаходження. Ми перенесемо логіку представлення даних в **Presenter**.

Для цього додамо рядок коду у метод `configurePickers()`:

```
func configurePickers() {
     shippingMethodTextField.inputView = shippingMethodPicker
     expirationDateTextField.inputView = expirationDatePicker
}
```

Потім змінимо метод *expirationDatePickerValueChanged():*

```
@IBAction func expirationDatePickerValueChanged(sender:
AnyObject) {
     let date = expirationDatePicker.date
     let request =
CreateOrder_FormatExpirationDate_Request(date: date)
     output.formatExpirationDate(request)
}
```

Після того, як користувач за допомогою елемента управління `UIPickerView` вибере значення для поля `Expiration date`, виконається метод `expirationDatePickerValueChanged()`. Результатом його роботи стане дата, яку ми розмістимо всередині конструкції під назвою `CreateOrder_FormatExpirationDate_request`. Це звичайна **Swift** структура, ініційована в файлі `CreateOrderModels.swift`. Вона складається з одного елемента даних `Date` типу `NSDate`:

```
struct CreateOrder_FormatExpirationDate_Request {
     var date: NSDate
}
```

Ви можете запитати, чому я використовую в назві структури символ “_”. У мовах програмування **Objective-C** і **Swift** для іменування класів, структур і перерахувань прийнято використовувати “горбатий регістр”. Дозвольте мені пояснити свою позицію.

`Create Order` — це ім'я екрану (сцени).

`FormatExpirationDate` — це намір або бізнес-правило. Запит вказує на межу між компонентами **`View Controller`** і **`Interactor`**. Тому ми отримуємо наступне правило для іменування: `CreateOrder_FormatExpirationDate_Request`.

Зверніть увагу на зручність читання і смислове навантаження у випадку з `CreateOrder_FormatExpirationDate_Request` і `CreateOrderFormatExpirationDateRequest`. Який з них говорить нам про сцену, наміри і межі найбільш виразно? Саме тому в моєму випадку вимоги ясності взяли гору над догмою.

Тепер повернімося до методу `expirationDatePickerValueChanged()`.

Після того, як ми створили `об'єкт-запит (request)` і ініціалізували його зі значенням дати від `UIPickerView`, нам потрібно просто викликати метод `output.formatExpirationDate(request)` для форматування вихідного значення `Expiration date`.

Так, то була правильна здогадка. Нам потрібно модифікувати протоколи `CreateOrderViewControllerOutput` і `CreateOrderInteractorInput`:

CreateOrderViewController:

```
protocol CreateOrderViewControllerOutput {
    var shippingMethods: [String] { get }
    func formatExpirationDate(request:
CreateOrder_FormatExpirationDate_Request)
}
```

CreateOrderInteractor:

```
protocol CreateOrderInteractorInput {
     var shippingMethods: [String] { get }
     func formatExpirationDate(request:
CreateOrder_FormatExpirationDate_Request)
}

class CreateOrderInteractor: CreateOrderInteractorInput {
     // MARK: Expiration date
     func formatExpirationDate(request:
CreateOrder_FormatExpirationDate_Request) {
          let response =
CreateOrder_FormatExpirationDate_Response(date: request.date)
          output.presentExpirationDate(response)
     }
}
```

Реалізація логіки представлення Expiration Date за допомогою компонента Presenter

У методі `formatExpirationDate()` ми створюємо об’єкт-відповідь (`response`) `CreateOrder_FormatExpirationDate_Response` як того вимагає наступне визначення у файлі `CreateOrderModels.swift`:

```
struct CreateOrder_FormatExpirationDate_Response {
     var date: NSDate
}
```

Ми ініціюємо об’єкт-відповідь (response) з датою, яку ми можемо отримати з об’єкту-запиту (request). **Interactor** нічого з нею не робить — він просто передає її в якості вхідного параметра у метод output.presentExpirationDate(response). Покищо ми не використовуємо в роботі бізнес-логіку, пов’язану з датою expiration date. Пізніше ми додамо одне бізнес-правило, коли виконаємо перевірку на відповідність вимогам expiration date.

Продовжимо додавання методу presentExpirationDate() у CreateOrderInteractorOutput і протоколи:

CreateOrderInteractor:

```
protocol CreateOrderInteractorOutput {
    func presentExpirationDate(response:
CreateOrder_FormatExpirationDate_Response)
}
```

CreateOrderPresenter:

```
protocol CreateOrderPresenterInput {
    func presentExpirationDate(response:
CreateOrder_FormatExpirationDate_Response)
}

class CreateOrderPresenter: CreateOrderPresenterInput {
    weak var output: CreateOrderPresenterOutput!

let dateFormatter: NSDateFormatter = {
        let dateFormatter = NSDateFormatter()
        dateFormatter.dateStyle = .ShortStyle
        dateFormatter.timeStyle =
NSDateFormatterStyle.NoStyle
```

```
        return dateFormatter
    }()

    // MARK: Expiration date
    func presentExpirationDate(response:
CreateOrder_FormatExpirationDate_Response) {
        let date =
dateFormatter.stringFromDate(response.date)
        let viewModel =
CreateOrder_FormatExpirationDate_ViewModel(date: date)
        output.displayExpirationDate(viewModel)
    }
}
```

У **CreateOrderPresenter** ми визначаємо константу NSDateFormatter. Метод presentExpirationDate() перетворює значення expiration date з типу NSDate в тип String. І вже строкове представлення дати записується в структуру CreateOrder_FormatExpirationDate_ViewModel, задану у файлі CreateOrderModels.swift.

Зверніть увагу, що дата в моделі представлення даних (view model) має тип String, а не NSDate. Робота компонента **Presenter** полягає в перетворенні отриманих даних у формат, придатний для демонстрації користувачу.

Оскільки для відображення дати інтерфейс використовує елемент UITextField, ми зобов'язані відформатувати дату в строковий літерал. Як правило, дані в моделі представлення є строковими або числовими.

```
struct CreateOrder_FormatExpirationDate_ViewModel {
     var date: String
}
```

Нарешті ми в змозі наказати **контролеру** відобразити наші дані, виконавши метод `output.displayExpirationDate(viewModel)`.

Це також означає, що нам необхідно реалізувати метод `displayExpirationDate()` в протоколах `CreateOrderPresenterOutput` і `CreateOrderViewControllerInput`:

Протокол CreateOrderPresenter:

```
protocol CreateOrderPresenterOutput: class {
     func displayExpirationDate(viewModel:
CreateOrder_FormatExpirationDate_ViewModel)
}
```

Протокол CreateOrderViewController:

```
protocol CreateOrderViewControllerInput {
     func displayExpirationDate(viewModel:
CreateOrder_FormatExpirationDate_ViewModel)
}

class CreateOrderViewController: UITableViewController,
CreateOrderViewControllerInput, UITextFieldDelegate,
UIPickerViewDataSource, UIPickerViewDelegate {
     // MARK: Expiration date
     func displayExpirationDate(viewModel:
CreateOrder_FormatExpirationDate_ViewModel) {
          let date = viewModel.date
          expirationDateTextField.text = date } }
```

Всередині методу `displayExpirationDate()` ми повинні взяти строкове значення дати з моделі представлення (view model) і привласнити його елементу `textField`:

```
expirationDateTextField.text = date.
```

Це все. Розгляд **VIP-циклу** закінчено.

Вихідний код прикладу доступний за посиланням:
https://github.com/Clean-Swift/CleanStore.

Ми отримали робочу форму для створення Замовлення. Користувач може вводити текст в текстові поля, а також вибирати способи доставки і терміни придатності за допомогою відповідних пікерів. Ваші бізнес-логіка і логіка представлення винесені за межі **View Controller** в компоненти **Interactor** і **Presenter**. Використання моделі структур користувача призначене для замикання компонентів **Clean Swift** у власних межах.

У цьому прикладі ми поки що не розглянули роботу **Worker** і **Router**. Коли ваша бізнес-логіка стає складнішою на допомогу приходить принцип поділу праці. Робота **Interactor** розбивається на декілька дрібніших завдань, які можуть виконуватися окремими **Workers**. Коли потрібно показати іншу сцену відразу після створення нового замовлення вам доведеться вдатися до послуг **Router** (маршрутизатора) для відокремлення логіки навігації.

Давайте пригадаємо, про що ми сьогодні дізналися:

- проблема **Massive View Controller** існує;
- **MVC** не є оптимальною архітектурою iOS додатків;
- дизайн–шаблони і рефакторинг стосуються більше технологій програмування, аніж архітектури додатків;
- тестувати можна лише додатки з правильною архітектурою;
- правильна архітектура робить легким процес внесення змін у код;
- код потрібно організовувати під задані процеси.

Крім того, ми познайомилися з архітектурою **Clean Swift** і **VIP–циклом**, розділили дані і бізнес-логіку для практичного кейса “Створення замовлення” і застосували **Clean Swift** для реалізації практичного кейса.

Ще раз згадаємо про **VIP–цикл**:

- компонент **View Controller** реагує на дії користувача, створює об’єкт–запит (request model) і відправляє його в **Interactor**;
- компонент **Interactor** обробляє об’єкт–запит (request model), створює об’єкт–відповідь (response model) і відправляє його в **Presenter**;
- компонент **Presenter** форматує дані з об’єкта–відповіді (response model), заповнює ними об’єкт–модель представлення даних (view model) і відправляє його назад у **View Controller**;
- компонент **View Controller** показує користувачу результати з моделі представлення даних (view model).

У майбутніх статтях ми продовжимо вивчення цього практичного кейса. Ви дізнаєтеся:

- як реалізувати перемикання в режим автоматичного заповнення поля `billing address` значенням із поля `shipping address`;
- як застосовувати **`Workers`** для перевірки полів форми створення нового замовлення;
- як зберігати дані нового замовлення, використовуючи **`Core Data`** під час натискання користувачем кнопки `Done`;
- як використовувати **`Router`** для навігації поміж сценами зі `Storyboards`;
- як застосовувати **`TDD`**-тестування.

Усі файли з текстами програми доступні для скачування на ресурсі `GitHub`.