НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

ННК "Інститут прикладного системного аналізу"

(повна назва інституту/факультету)

Системного проектування

(повна назва кафедри)

«На правах рукопису» УДК 004:004.453

«До захисту допущено»

Завідувач кафедри

_____ Мухін В.Є.

(підпис) (ініціали, прізвище)

"_____" _____ 2022 р.

Магістерська дисертація

зі спеціальності (спеціалізації) 122 – комп'ютерні науки та інформаційні

(код і назва спеціальності)

технології (Системне проектування сервісів)

на тему: Розробка комбінованої архітектури мобільних додатків на основі аналізу архітектур MV(x) та Unidirectional Data Flow

Виконав (-ла): студент (-ка) 6 курсу, групи ДА-11мп

(шифр групи)

Зарічний Ярослав Сергійович

(прізвище, ім'я, по батькові) (підпис)

Науковий керівник ас. Яременко В. С

(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант к.т.н., доцент Гіоргізова-Гай В. Ш..

Консультант Розробка стартап-проекту ас. Яременко В. С.

(назва розділу) (науковий ступінь, вчене звання, , прізвище, ініціали) (підпис)

Рецензент к.т.н., доцент Недашківська Н. І.

(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2022 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут/факультет ННК "Інститут прикладного системного аналізу"

(повна назва)

Кафедра Системного проектування

(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною (освітньо- науковою) програмою

Спеціальність (спеціалізація) 122 – комп'ютерні науки та інформаційні технології (Системне проектування сервісів)

(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Мухін В.Є.
(підпис) (ініціали, прізвище)
«__»_____2022 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Зарічному Ярославу Сергійовичу

(прізвище, ім'я, по батькові)

1. Тема дисертації Розробка комбінованої архітектури мобільних додатків на основі аналізу архітектур MV(x) та Unidirectional Data Flow
науковий керівник дисертації ас, Яременко В. С,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «___»_____20_____р. №_____

2. Строк подання студентом дисертації

3. Об’єкт дослідження «Побудова клієнтських шаблонів проектування при розробці мобільних додатків»

4. Предмет дослідження (Вихідні дані – для магістерської дисертації за освітньо-професійною програмою) «Проектування та розробка комбінованої клієнтської архітектури для мобільних додатків на основі існуючих шаблонів MV(X) та Unidirectional Data Flow для платформи iOS»

5. Перелік завдань, які потрібно розробити

1. Провести аналіз нових тенденцій, проблем і вимог до архітектури, які виникають у сучасних розробників мобільних додатків;
2. Провести порівняльний аналіз сучасних архітектурних моделей для побудови мобільних додатків.
3. Визначити параметри архітектур та способи їх покращення.
4. Запропонувати модифіковану модель, яка зможе поліпшити та модифікувати вимоги проектів щодо можливостей модульного тестування, продуктивності (швидкодії), впровадження залежностей, забезпечення масштабованості та модульність даної архітектури.

6. Орієнтовний перелік графічного (ілюстративного)

матеріалу презентація на тему “Комбінована архітектура для мобільних застосунків ”

7. Орієнтовний перелік публікацій

Зарічний Я. С. Комбінована архітектура для мобільних застосунків / Я. С. Зарічний, В. Ш. Гіоргізова-Гай, В. С. Яременко. // І Всеукраїнська науково-практична конференція «Системні науки та інформатика» Збірник доповідей. – 2022. – С. 315–322.

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Розробка стартап-проекту	Яременко В. С.		

Комбінована архітектура для мобільних застосунків	Гіоргізова-Гай В. Ш.		

9. Дата видачі завдання
27.06.2022

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів магістерської дисертації	Примітка
1	2	3	4
1	Звіт з проходженя практики	25 жовтня 2022	
2	Дослідження проблематики у дисертації	10 листопада 2022	
3	Аналіз даних та написання дисертації	30 листопада 2022	
4	Документування та редагування результатів роботи	10 грудня 2022	

Студент _____ Зарічний Я. С. (підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ ас. Яременко В. С.

РЕФЕРАТ

магістерської дисертації Зарічного Ярослава Сергійовича на тему «Розробка комбінованої архітектури мобільних додатків на основі аналізу архітектур MV(x) та Unidirectional Data Flow»

Робота виконана на 84 сторінках, містить 29 ілюстрацій, 23 таблиць. При підготовці використовувалась література з 22 джерел.

Актуальність теми дослідження полягає в тому, що побудова правильно спроектованої архітектури надає компаніям можливість швидше вирішувати проблеми користувачів, створювати нові модулі функціональності та поєднувати ці модулі вже з існуючими. Це дозволяє заощаджувати кошти на розробку програмного забезпечення, а також проектувати більш стабільні інформаційні системи.

Мета та задачі дослідження. Метою даної роботи є дослідження існуючих архітектурних рішень для побудови мобільних застосунків, їх переваг та недоліків, а також доцільності використання.

Об'єкт досліджень - Побудова кліентських шаблонів проектування при розробці мобільних додатків.

Предмет досліджень - Проектування та розробка комбінованої клієнтської архітектури для мобільних додатків на основі існуючих шаблонів MV(X) та Unidirectional Data Flow для платформи iOS.

Методи досліджень. Для вирішення проблеми в даній роботі використовуються методи аналізу і синтезу, системного аналізу, порівняння, логічного узагальнення результатів, проектування логічних структур даних.

Наукова новизна. Наукова новизна дипломної роботи полягає у розробленій архітектурній моделі, яка надає можливість комбінувати існуючі архітектурні моделі для використання певної в залежності від технічних умов і задач бізнесу. Також дана комбінована модель

відповідає основним принципам і умовам до побудови програмного забезпечення. В додаток до комбінованої моделі було створено декларативний фреймворк для побудови графічного відображення, який надає усі переваги декларативної парадигми програмування.

Потенційні застосування та практична цінність результатів дипломної роботи:

1) Розроблену архітектурну модель можливо застосувати як для нових реалізацій мобільних застосунків так і для покращення існуючих.
2) Розроблений фреймворк для побудови графічного відображення є доволі самостійним, тому його можливо використовувати на будь-яких проектах.

Апробація результатів дисертації. Результати дослідження опубліковано на І Всеукраїнській науково-практичній конференції «Системні науки та інформатика».

Публікації:

Зарічний Я. С. Комбінована архітектура для мобільних застосунків / Я. С. Зарічний, В. Ш. Гіоргізова-Гай, В. С. Яременко. // І Всеукраїнська науково-практична конференція «Системні науки та інформатика» Збірник доповідей. – 2022. – С. 315–322.

Ключові слова: Комбінована архітектура, Чиста архітектура, Двоспрямовані архітектури, Односпрямовані архітектури, Model-View-Controller, Model-View-Presenter, Model-View-ViewModel, Unidirectional Data Flow.

ABSTRACT

Zarichny Yaroslav Serhiyevich's master's thesis on the topic «Development of a combined mobile application architecture based on the analysis of MV(x) and Unidirectional Data Flow architectures»

The work is completed on 84 pages, contains 9 illustrations, 26 tables. Literature from 18 sources was used in the preparation.

The relevance of the research topic is that the construction of a properly designed architecture provides companies with the opportunity to solve user problems faster, create new modules of functionality and combine these modules with existing ones. This allows you to save money on software development, as well as design more stable information systems.

The purpose and objectives of the research. The purpose of this work is to study existing architectural solutions for building mobile applications, their advantages and disadvantages, as well as feasibility of use.

Object of research - Construction of client design templates in the development of mobile applications.

Research subject - Design and development of a combined client architecture for mobile applications based on existing MV(X) and Unidirectional Data Flow patterns for the iOS platform.

Research methods. To solve the problem in this paper, methods of analysis and synthesis, system analysis, comparison, logical generalization of results, and design of logical data structures are used.

Scientific novelty. The scientific novelty of the thesis consists in the developed architectural model, which provides an opportunity to combine existing architectural models for the use of certain depending on the technical conditions and tasks of the business. Also, this combined model corresponds to the basic principles and conditions for building software. In addition to this model, a declarative framework was created for building a graphical display,

which provides all the advantages of a declarative programming paradigm. Potential applications and practical value of the results of the thesis:

1) The developed architectural model can be applied both for new applications of mobile applications and for the improvement of existing ones;

2) The developed framework for building a graphic display is quite independent, so it can be used on any projects.

Approbation of the results of the dissertation. The results of the research were published at the 1st All-Ukrainian Scientific and Practical Conference "System Sciences and Informatics".

Publications: Zarichnyi Y. S. Combined architecture for mobile applications / Y. S. Zarichnyi, V. Sh. Giorgizova-Gai, V. S. Yaremenko. // 1st All-Ukrainian Scientific and Practical Conference "System Sciences and Informatics" Collection of reports. – 2022. – P. 315–322.

Keywords: Combined architecture, Pure architecture, Bidirectional architectures, Unidirectional architectures, Model-View-Controller, Model-View-Presenter, Model-View-ViewModel, Unidirectional Data Flow.

ЗМІСТ

ВСТУП

Розробка масштабованих і стійких додатків є складною технічною задачею, в якій необхідно врахувати не лише базові вимоги до продукту, а й закласти можливості до розширення функціоналу, масштабованості тощо. Однак, коли справа доходить до мобільних програм або платформ, завжди існувала проблема з чистою архітектурою [1]. У випадку з Android це може бути тому, що Google не підтримує та навіть не рекомендує жодної конкретної архітектури. Apple, з іншого боку, запропонувала архітектуру MVC для UIKit, але ця пропозиція викликала багато суперечок, і багато експертів стверджували, що це не дуже гарне рішення. В роботі було досліджено два типи архітектури: односпрямовані, двоспрямовані та загальні принципи чистого програмування та побудови архітектур. До односпрямованих належать UDF побідні архітектури, до двоспрямованих MVX подібні та Чиста архітектура. Базуючись на перевагах та недоліків кожного підходу було запропоновано побудувати комбіновану архітектуру, яка надає можливість комбінувати переваги кожного типу архітектур та слідувати загальним принципам побудови чистої архітектури. Наступним кроком було розроблено декларативний фреймворк для побудови відображення з застосуванням основних принципів проектування програмного забезпечення, кодогенерування.

1 АНАЛІЗ АРХІТЕКТУРНИХ МОДЕЛЕЙ ЯК АРХІТЕКТУРНИХ ШАБЛОНІВ ДЛЯ iOS

1.1 ШАБЛОН MODEL VIEW CONTROLLER ДЛЯ iOS

MVC (Model View Controller) — це програмний шаблон для реалізації інтерфейсів користувача на електронних пристроях. За словами його архітектора Трюгве Реенскауга, «MVC було створено як рішення загальної проблеми надання користувачам контролю над своєю інформацією з різних точок зору». Трюгве задумав MVC, працюючи запрошеним науковцем у дослідницькій лабораторії Xerox Palo Alto Research Laboratory (PARC) з літа 1978 по 1979 рік. [1] Під час роботи в PARC він зосередився на підтримці LRG (навчальної дослідницької групи), команди, яка створювала DynaBook. (Рис. 2.1)

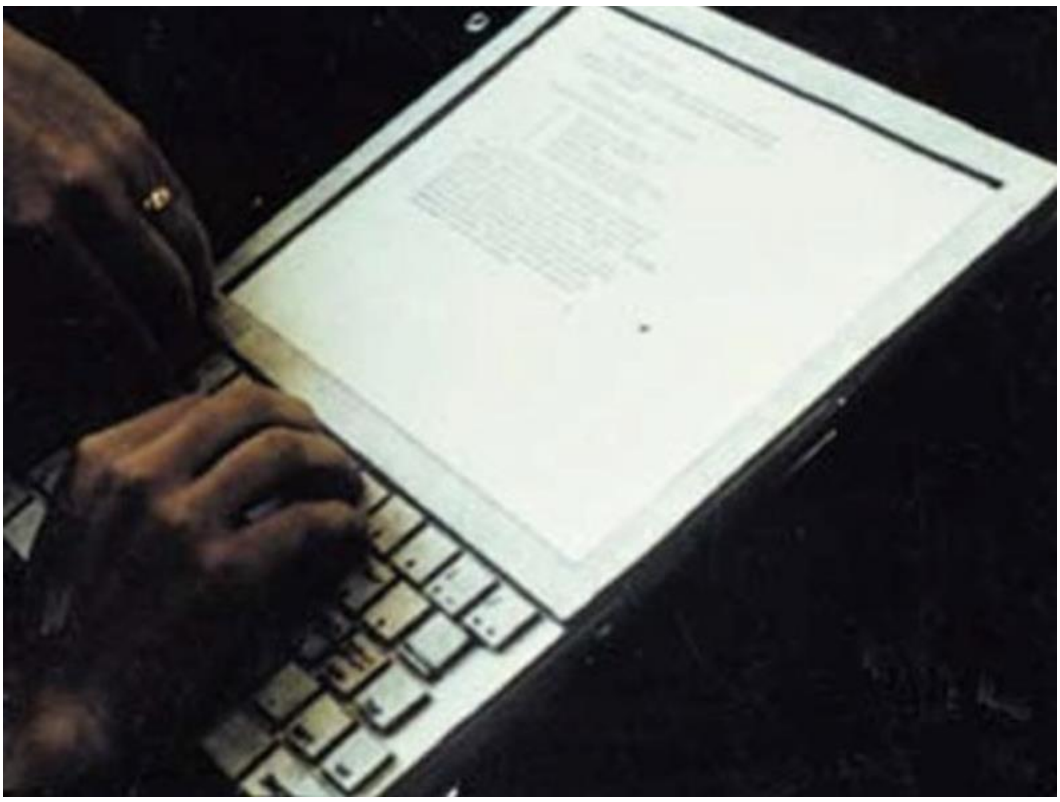


Рисунок 2.1 – DynaBook [1]

DynaBook був першою концепцією того, що тепер стало відомим як планшетний ПК. Його творці передбачили загальний інструмент, який подолає прірву між користувачем та інформацією, яка є для нього актуальною. Ранні концепції були зроблені з урахуванням дітей, DynaBook мав служити навчальним комп'ютером для дітей різного віку. Таким чином, він повинен бути простим у використанні, він повинен бути інтуїтивно зрозумілим. Як описують творці, «Користувач був царем; все, що робилося в LRG, було зроблено для його підтримки». [1]

Саме на цьому тлі було створено MVC як допоміжну архітектуру програмного забезпечення для нового типу персональних комп'ютерів. Комп'ютер, який ставить на перше місце зручність використання без шкоди для функцій. Це мислення є очевидним у початковій теорії MVC. Як каже Трюгве, «основна мета MVC — подолати розрив між розумовою моделлю людини та цифровою моделлю, яка існує в комп'ютері. Ідеальне рішення MVC підтримує у користувача ілюзію безпосереднього перегляду та маніпулювання інформацією домену. [1] Структура корисна, якщо користувачеві потрібно побачити той самий елемент моделі одночасно в різних контекстах і/або з різних точок зору». (Рис. 2.2)

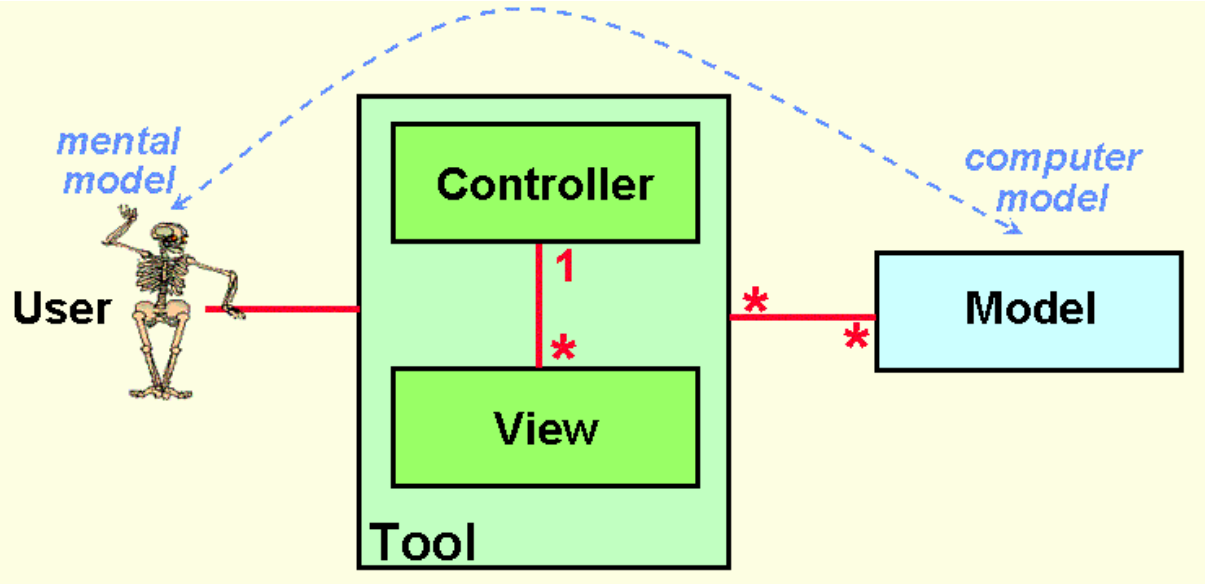


Рисунок 2.2 - Оригінальна модель MVC від Trygve Reenskaug [1]

Перша реалізація MVC була виконана групою LRG на мові під назвою smalltalk. smalltalk є однією з перших справжніх об'єктно-орієнтованих мов. Сьогодні MVC є архітектурним шаблоном вибору для настільних GUI, веб-додатків і навіть мобільних додатків. Він також став шаблоном вибору для найпопулярніших веб-фреймворків.

У початковій формулюванні кожне представлення відображало представлення всього або частини одного об’єкта домену, який міг бути складеним (і який зазвичай мав Model як суперклас, щоб отримати необхідні можливості сповіщення про зміни). Хоча намір полягав у підтримці графічних інтерфейсів для додатків, розроблених у Smalltalk, інструменти середовища Smalltalk – зокрема браузери – також були реалізовані з використанням парадигми MVC. Браузери були трохи дивними, оскільки браузер моделі був об’єктом браузера, який, по суті, замінював класи зображення, методи, категорії класів і протоколи методів, представлені на різних панелях браузера. [1]

Розмірковуючи над досвідом Smalltalk-80 із браузерами та іншими інтерфейсами MVC, розробники ParcPlace Systems (утворені членами PARC Learning Research Group, які брали участь у початковій розробці Smalltalk під керівництвом Адель Голдберг) переробили системні інструменти для свого покоління Smalltalk ObjectWorks. продукти [близько 1987 року -- знайдіть фактичну дату]. Вони зрозуміли, що розробники інструментів і додатків написали велику кількість класів перегляду тексту та списків, тоді як реалізації відрізнялися не стільки тим, як вони відображають текст і списки, скільки конкретними деталями загальних тем, таких як:

які команди з’являються в меню середньої кнопки («жовтий» у Smalltalk-80 і раніше, «<operate>» у VisualWorks) та методи їх реалізації

яке повідомлення надіслати моделі, щоб отримати відображені дані або зберегти нове значення (передбачаючи AspectAdapters VisualWorks).

Вони розробили підключення відображення — загальні текстові та спискові відображення, деталі яких вказувалися в методах із багатьма аргументами, що використовуються для їх створення. Відображення, що підключаються, значно зменшили кількість окремих класів представлень і контролерів, необхідних у системі та типових програмах, оскільки дуже багато інтерфейсів складалися в основному або повністю з панелей тексту та списків. [1]

Коли Apple представила свій iPhone SDK у 2008, вони рекомендували створили посібник та рекомендували дотримуватися вказівок в ньому. В цьому посібнику також було запропоновано використовувати архітектурний шаблон MVC, котрий був спочатково представлен компанією SmallTalk [1]. Ця компанія описувала цей шаблон як: “шаблон високого рівня, оскільки він стосується глобальної архітектури програми”. [2] З технічного огляду ця архітектура складається з комбінації класичних шаблонів «Банди Чотирьох» [3] та ділить побудови програми на три головних компонента: відображення, контроллера та моделі. (Рис. 2.3)

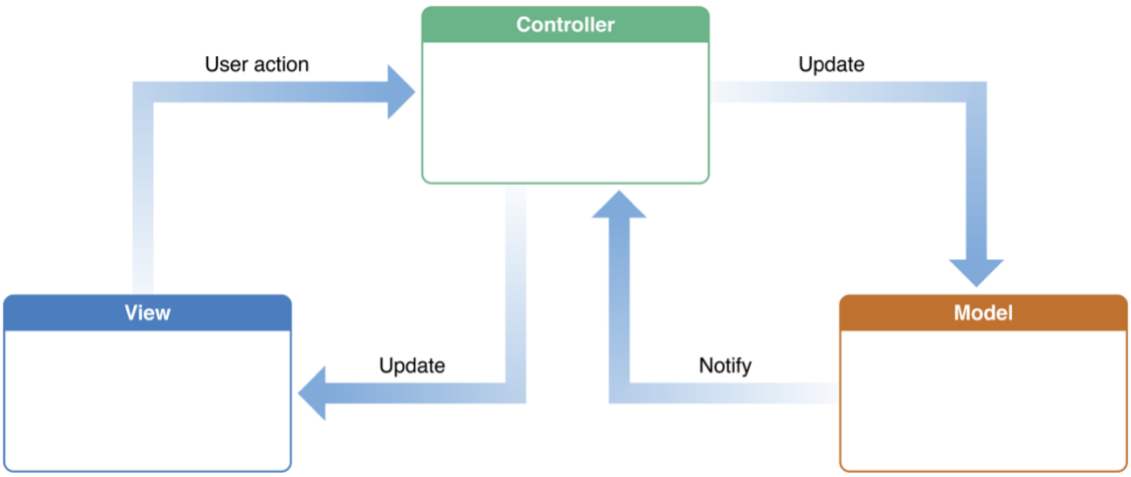


Рисунок 2.3 – Apple MVC [2]

MVC спочатку мав на меті розподілити компоненти програми між окремими частинами. Але загалом результат був такий:

Відсутність розподілу: контролер зрештою виконує всю роботу. від обробки взаємодії користувачів до налаштування переглядів. здійснення мережевих дзвінків, аналіз даних і так далі. Ця пробелма також відома, як Massive View Controller.

Низьке охоплення тестуванням: окрім порушення принципу єдиної відповідальності. Контролер тісно пов’язаний із життєвим циклом перегляду. Тестування контролерів перегляду стає важким завданням.

Перші приклади для розробників для iPhone SDK мали у собі приклади цього шаблону: клас UIView працював як відображення, AppDelegate як контролер усього застосунку, а прикладів щодо моделей не було. В цей час, перші архітектурні патерни для ОС Android були дуже схожими, відображення будувалось за допомогою XML файлів, а клас Activity був у ролі контролера. Також і в цьому випадку теж існувала проблема - відсутність прикладів класів моделей. Ця проблема мала своє відображення на багатьох мобільних застосунків, оскільки, призводить до збільшення класів контролерів.

1.2 ШАБЛОН MODEL VIEW PRESENTER ДЛЯ iOS

Model–View–Presenter (MVP) є похідним архітектурного шаблону Model–View–Controller (MVC) і використовується здебільшого для створення інтерфейсів користувача. [4]

Шаблон MVP дуже схожий на шаблон MVC. У цій архітектурі шаблон «P» означає презентатора. Таким чином, він має такі частини, як MVC, але тут контролер замінено презентатором, хоча вони не виконують ту саму функцію. Презентатор може адресувати всі події інтерфейсу користувача від імені представлення. Він приймає дані від користувачів, обробляє дані в моделі, а потім перетворює результати назад у представлення. (Рис. 2.4) [5].

MVP також відомий як складений шаблон. Шаблон MVP зазвичай виконується з додатками Windows Forms і ASP.NET WebForms.

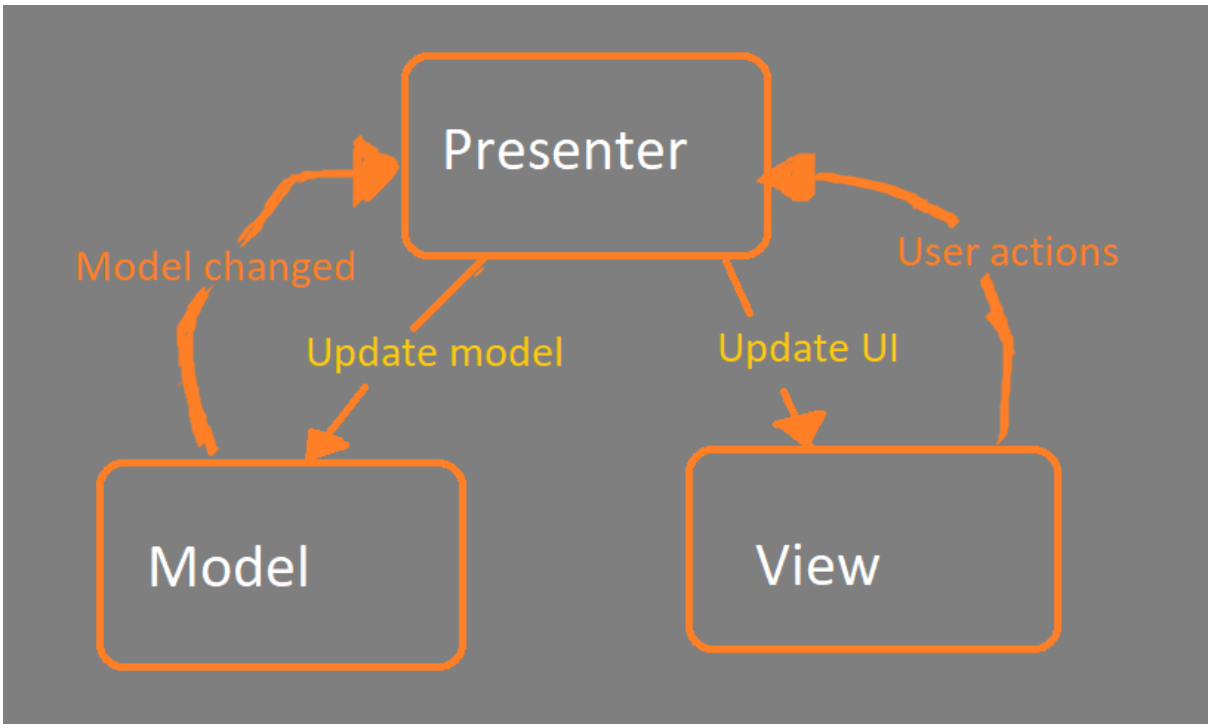


Рисунок 2.4 – Компоненти MVP [5]

Архітектура MVP покращує вище згадані проблеми з MVC. шляхом додавання основного компонента, яким є Presenter. На перший погляд, це виглядає як MVC, але з ключовою відмінністю:

Тепер ViewController розглядається як представлення. Це означає, що він включатиме лише пов'язаний з переглядом код, нічого більше. і вся логіка буде реалізована в Presenter [6] .

Тоді опис компонентів виглядає наступним чином:

View: тепер представлення складається з представлень і контролерів перегляду з усіма налаштуваннями інтерфейсу користувача та подіями.

Presenter: цей компонент відповідатиме за всю логіку, включаючи реагування на дії користувача та оновлення інтерфейсу користувача (через делегата). і найголовніше те, що наш презентер не буде залежати від UIKit. що означає добре ізольований, тому його легко перевірити;)

Model: роль моделі буде точно такою ж як у MVC.

Важливо зазначити, що MVP використовує пасивний шаблон перегляду. це означає, що всі дії будуть передані ведучому. Що запускатиме оновлення інтерфейсу за допомогою делегатів. тому перегляд лише пропускатиме дії та слухатиме оновлення доповідача.

Ступінь логіки, яку дозволено реалізовувати у відображенні, залежить від конкретних реалізацій. З одного боку, представлення може бути повністю пасивним, пересилаючи всі операції взаємодії до презентатора. У цьому формулюванні, коли користувач запускає метод події перегляду, він не робить нічого, крім виклику методу презентатора, який не має параметрів і не повертає значення. Потім презентатор отримує дані з подання за допомогою методів, визначених інтерфейсом відображення. Нарешті презентатор оперує моделлю та оновлює відображення результатами операції. Інші версії MVP дозволяють певну

свободу щодо того, який клас обробляє певну взаємодію, подію чи команду. Це часто більше підходить для веб-архітектур, де перегляд, який виконується в браузері клієнта, може бути найкращим місцем для обробки певної взаємодії або команди. [5]

З точки зору шарів, клас Presenter можна вважати таким, що належить до прикладного рівня в системі з багаторівневою архітектурою, але його також можна розглядати як власний рівень презентатора між рівнем програми та рівнем інтерфейсу користувача.

1.3 ШАБЛОН MODEL VIEWMODEL MODEL ДЛЯ iOS

Шаблон проектування Model-View-ViewModel (MVVM) привертає багато уваги (як і має бути). Те місце, де багато розробників застрягають (особливо розробники, які тільки підходять до світу XAML), – це реалізація. Часто розробники підходять до фреймворку/набору інструментів, який пропонує допомогу з впровадженням шаблону MVVM, і вони загрузають у тонкощах конкретного фреймворку, не розуміючи самого шаблону [7].

Шаблон MVVM базується на концепціях попередніх шаблонів проектування презентації, включаючи MVC (Model-View-Controller) і MVP (Model-View-Presenter). Шаблон MVC був розроблений до інструментів RAD, які використовуються сьогодні, коли розробнику доводилося створювати елементи інтерфейсу користувача з нуля (або залучати їх із спільної бібліотеки). Ендрю Хант і Девід Томас описують ключову концепцію MVC як «відокремлення моделі як від графічного інтерфейсу користувача, який її представляє, так і від елементів керування, які керують представленням» [7]. Ці шаблони перейшли в наші сучасні інструменти (як у ASP.NET MVC). Як можливо зрозуміти з назв, ці шаблони поділяють спільні ідеї: модель і вигляд.

Шаблон MVVM — це варіант шаблонів MVC і MVP на основі моделі представлення XAML. XAML пропонує дуже багатий механізм зв’язування, і шаблон MVVM використовує цей механізм зв’язування як «біндінг», який утримує все разом (Рис. 2.5).

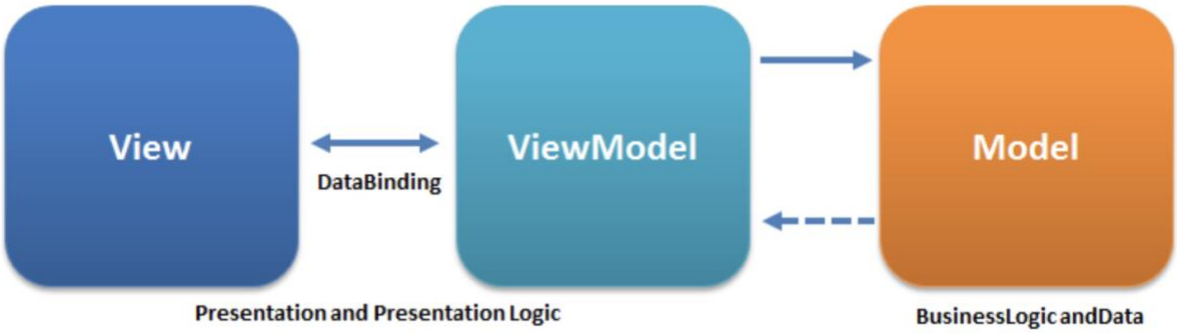


Рисунок 2.5 – Компоненти MVVM [7]

Як випливає з назви шаблону, шаблон MVVM складається з трьох основних компонентів. Розглянемо кожен із компонентів і побачимо як вони взаємодіють один з одним.

Модель

Модель інкапсулює бізнес-логіку та дані, які використовуються програмою. Найкращий спосіб подумати про модель – це все, що не є презентацією – при використанні підходу багаторівневої програми це все, що знаходиться нижче рівня презентації (який може включати бізнес-рівень, рівень обслуговування, рівень доступу до даних, рівень зберігання даних тощо). У більшості сценаріїв наша модель буде розділена на різні компоненти. Наприклад, ми можемо мати репозиторій, який відповідає за отримання та збереження об’єктів через службу, а служба відповідає за взаємодію зі сховищем даних (наприклад, базою даних). І які типи об’єктів даних будуть представлені, будуть різними; вони можуть бути смарт-об’єктами (якщо ми робимо об’єктно-орієнтоване програмування) або об’єктами даних (якщо ми робимо розробку на основі сутностей або доменів) [6].

Фактична реалізація моделі не є важливою для загального шаблону MVVM, хоча реалізація об’єктів у моделі може впливати на те, як об’єкти

відображаються у ViewModel. Модель не повинна мати жодної логіки представлення.

ViewModel

ViewModel інкапсулює логіку представлення та дані, які використовуються View. Ми можемо розглядати це як «слиз», яка тримає речі разом. MVVM розроблено на основі потужного механізму зв'язування даних, доступного в XAML. ViewModel відповідає за те, щоб об'єкти даних, доступні в моделі, були доступні як властивості, до яких View може прив'язувати дані. Але ViewModel відповідає не тільки за надання об'єктів моделі як властивостей. Він також надає властивості, які використовуються для логіки представлення програми. Наприклад, припустимо, що у нас є об'єкт Order, і чи буде він доступним лише для читання, залежатиме від того, чи замовлення вже було відправлено. Модель оброблятиме визначення стану лише для читання об'єкта (оскільки це частина бізнес-логіки), але тоді ViewModel може надавати властивість OrderReadOnly, яку може використовувати View. Потім View може прив'язати властивість IsReadOnly для певного елемента керування (або набору елементів керування) до властивості OrderReadOnly ViewModel [6].

ViewModel також обробляє будь-яку взаємодію між View і Model. Це досягається шляхом показу методів або команд, які View може викликати. Потім ViewModel використовує ці методи, щоб викликати модель.

View

Перегляд — це те, що користувач бачить на екрані — усі елементи інтерфейсу користувача. Це набір елементів XAML, які складають

інтерфейс користувача (елементи керування користувача, кнопки, мітки, текстові поля), а також допоміжні елементи, які допомагають цим елементам (стилі, анімація, шаблони елементів керування).

Як можна побачити раніше, MVVM базується на прив'язці даних XAML. Це означає, що представлення зазвичай переповнене атрибутами, пов'язаними з даними. Як зазначено в описі ViewModel вище, це стосується не тільки даних, але й інших властивостей, які впливають на взаємодію. Наприклад, у нас може бути TextBox, властивість якого Text — це дані, прив'язані до властивості в ViewModel (наприклад, OrderDate). Але той самий TextBox також може мати дані властивості IsReadOnly або IsEnabled, прив'язані до властивостей у ViewModel. Ми навіть можемо мати такі візуальні властивості, як FontSize або Foreground, визначені прив'язкою даних до властивостей ViewModel [6].

Поділ інтересів і взаємодія

Тепер, коли ми побачили компоненти та за що вони відповідають, давайте подивимося, як вони взаємодіють один з одним.

Представлення знає лише про ViewModel. Представлення не має прямих знань про модель. Він знає лише про ViewModel. Представлення зазвичай відповідає за створення ViewModel, і це майже завжди взаємозв'язок один-до-одного. В ідеальному світі View не мав би коду (але деякі люди більш наполегливі щодо цього, ніж інші - це залежить від реалізації).

ViewModel знає про Модель. ViewModel несе відповідальність за отримання будь-яких об'єктів, які їй потрібно виставити з моделі. Це може бути зв'язок «один до одного» або «один до багатьох». Прикладом взаємозв'язку «один-до-одного» може бути ViewModel, який показує об'єкт Customer із моделі; прикладом зв'язку «один-до-багатьох» може бути ViewModel, яка потребує відображення Замовлення, яке містить

Клієнта разом із кількома Продуктами. Спосіб їх відображення (як окремі чи складені об’єкти) залежить від потреб програми. ViewModel має залежати від моделі бізнес-логіки (наприклад, виклик моделі, щоб визначити, чи можна редагувати об’єкт на основі значень даних об’єкта).

ViewModel не має прямих знань про View - це означає, що ViewModel не повинна посилатися на будь-які елементи керування або елементи інтерфейсу користувача, які є частиною View. Він може містити логіку представлення (наприклад, відображення та налаштування властивостей, які визначають, відображаються чи приховані елементи керування), але він повинен залишити на вибір View, щоб визначити, як елементи насправді представлені [6].

Модель не має прямих знань про ViewModel або View. Від ViewModel залежить будь-яка взаємодія з моделлю (ніколи не навпаки). Модель повинна містити бізнес-логіку, але логіка представлення має бути обмежена ViewModel.

Приклад: Взаємодія власності

Розглянемо конкретне бізнес-правило, яке впливає на інтерфейс користувача, і подивимося, які компоненти несуть відповідальність на цьому шляху. Повертаючись до нашого прикладу, скажімо, що замовлення не можна змінити після його відправлення. Модель містила б бізнес-логіку, щоб визначити, чи було відправлено замовлення (можливо, перевіривши поле ShippingDate), а потім належним чином установити властивість ReadOnly для об’єкта Order. ViewModel візьме цю властивість ReadOnly і представить її View. Представлення може визначити, як використовувати цю властивість за допомогою зв’язування даних. Наприклад, View може мати властивість IsReadOnly для набору даних TextBoxes, прив’язаних до властивості ViewModel, або він може повністю змінити візуальні елементи, показуючи TextBlocks, якщо для

властивості ReadOnly ViewModel встановлено значення true. Важливо пам’ятати, що модель відповідає за бізнес-логіку (визначає, чи є порядок змінним), ViewModel відповідає за надання цієї властивості View, а View відповідає за використання властивості для керування фактичним інтерфейсом користувача. елементів.

Приклад: Взаємодія методу

Тепер інша перспектива: скажімо, у нашому поданні є кнопка «Зберегти», і нам потрібно зберегти поточний об’єкт на екрані. Примітка: існує кілька способів реалізації цього; це лише один шлях. У нашому представленні ми маємо кнопку з даними властивості Command, прив’язаними до SaveOrderCommand у ViewModel. У ViewModel виконується об’єкт SaveOrderCommand, який викликає метод Repository<Order>.Save() (у моделі) і передає поточний об’єкт Order. У моделі метод Save у сховищі виконує відповідну дію, наприклад збереження замовлення в базі даних. Зверніть увагу на взаємодію: View звертається лише до ViewModel, і це залежить від ViewModel, щоб викликати модель. Потім модель виконує фактичну роботу.

MVVM: Проблемний простір

Тепер, коли побачили різні компоненти шаблону MVVM, подивимось, чому потрібно використовувати цей шаблон. За своєю суттю шаблон стосується трьох основних проблем:

Проблемний простір: поєднання презентації та бізнес-логіки

Розробники додатків часто змішують код логіки презентації та код бізнес-логіки. За допомогою наших сучасних інструментів зробити це надто просто: просто відкрийте новий проект WPF у Visual Studio, перетягніть деякі елементи керування в робочу область XAML, перейдіть до коду позаду та почніть створювати код разом. Але кодування таким чином призводить до кошмару обслуговування.

Рішення: MVVM заохочує якісне розділення презентації та бізнес-логіки. Завдяки чітко визначеним компонентам розробник має відповідні місця для розміщення логіки представлення (модель перегляду) окремо від бізнес-логіки (моделі).

Проблемний простір: модульне тестування рівня презентації є складним

Коли ми проводимо модульне тестування (а це має бути завжди), наша мета полягає в тому, щоб мати тести, які охоплюють якомога більше нашого коду - це включає в себе функції презентації. Під час розробки модульних тестів із традиційним інтерфейсом користувача (обробники подій у коді позаду) важко розділити окремі функції. Наприклад, якщо ми хочемо провести модульне тестування події натискання кнопки, ми можемо створити модульний тест навколо самого обробника події. Але в налаштуванні для тесту нам потрібно буде створити екземпляр об’єкта Button (параметр «відправник») за допомогою підробки або макету. Це також може означати, що нам потрібно створити екземпляр цілого об’єкта UserControl як батьківського для Button, щоб отримати дійсний тест [6].

Рішення: ViewModel у MVVM піддається чистішому модульному тестуванню. Використовуючи наведений вище приклад, припустимо, що замість того, щоб подія натискання кнопки безпосередньо виконувала дію, вона викликає метод у ViewModel. Тепер ми можемо легко тестувати метод у ViewModel. Оскільки ViewModel не має посилання на елементи інтерфейсу користувача, нам не потрібно було б створювати їх екземпляри для модульного тесту. Це призводить до коду, який легше перевірити та легше виділити. Примітка: виклики методів, як правило, переміщуватимуться від ViewModel до Model, але оскільки ми маємо ці

«шви» в нашій програмі, легше підробити або висміяти ці взаємодії в наших модульних тестах [6].

1.4 ШАБЛОН UNIDIRECTIONAL DATA FLOW ДЛЯ iOS

Основна ідея односпрямованих архітектури полягає в тому, щоб дані в додатку рухалися тільки в одному напрямку: від моделей додатків до інтерфейсу користувача, але не навпаки. Якщо в UI щось трапилось, він ніяк не намагається інтерпретувати ці події. Все, що робить UDF — відправляє події в модель, яка вирішує, як оновити стан системи (Рис. 2.6).

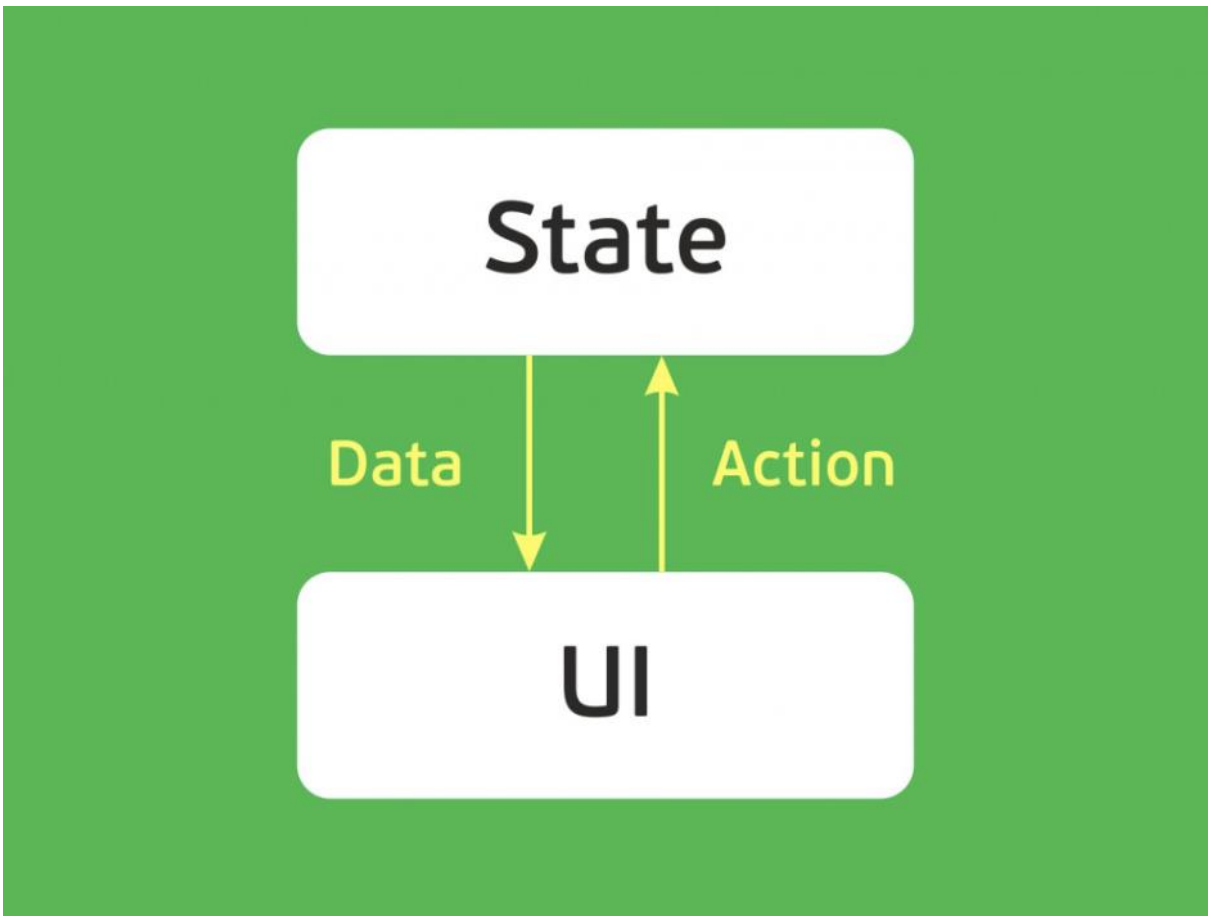


Рисунок 2.6 – Основний принцип UDF архітектур [7]

У такій схемі легко отримуємо те, що дані, передані в інтерфейс користувача, були іммутабельними. Інтерфейс користувача отримує на вході дані та відображає їх, а якщо потрібно що-небудь змінити, інтерфейс користувача надсилає подію (Action) у модель та очікує, коли до нього прийдуть уже оновлені дані [7].

Різні фреймворки по-різному реалізують моделі додатків. Спробуємо знайти в них загальні частини. Назви приводяться з реалізації Redux.

Стан (State) — стан системи. Це іммутабельні моделі, які описують поточний стан додатка.

Подія (Action) — події в системі. Допомагають через графічне відображення (UI) повідомити про виниклі зміни та повідомити про цю модель.

Перетворювач (Reducer) — зазвичай це чиста функція з сигнатурою (State, Action) -> State. Єдине місце, де дозволено змінювати Стан. При вході отримує старий Стан і виниклу Подію, і формує новий Стан. У деяких фреймворках є додаткові параметри або повертаються значення.

Склад (Store) — агрегующа сутність, яка зберігає в собі Стан і запускає перетворювання стану. У якості інтерфейсу надається можливість відправити Події і підписуватися на оновлення Стану. Чаще за все один на додаток.

В місці це працює так: (Рис. 2.7)

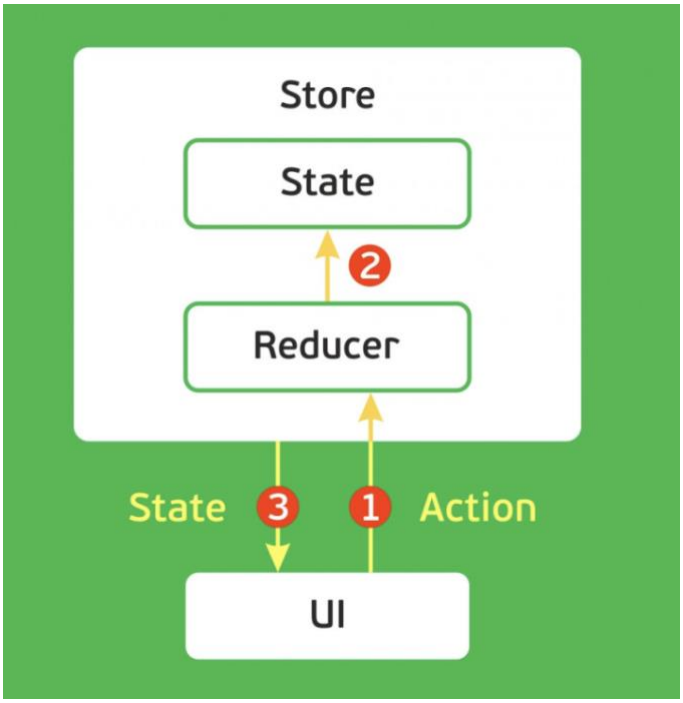


Рисунок 2.7 – Схема роботи UDF архітектур [7]

1. В інтерфейсі користувача сталася Подія, і вона відправляється до Складу.
2. Склад викликає Перетворювач і передає в якості параметрів поточний Стан і Подію, яка прийшла. На виході отримує — новий Стан, який зберігається в Склад замість старого.
3. Склад оповідає своїх підписників і передає їм оновлений Стан.

Може скластися враження, що такий підхід далекий від мобільної розробки і не підходить ні для iOS, ні для Android. Насправді Apple і Google використовують односпрямований потік даних у своїх фреймворках. Якщо уважно придивитися до схеми роботи SwiftUI, можливо побачити багато згожого з нашою схемою. Google прямим текстом згадує односпрямований потік даних у документації по Jetpack Compose.

Розглянемо плюси Unidirectional Data Flow:

- Чіткий поділ доменної логіки та сайд-ефектів. Принцип не новий і давно використовується у функціональному (чисті функції, монади) та об'єктно-орієнтованому програмуванні (CQRS). Однак більшість мобільних архітектур не акцентують увагу на тому, як реалізовувати модель програми, і бізнес-логіка часто просочується в Controller/Presenter/Interactor або View. UDF дає чіткі інструкції, як організувати доменний шар програми і отримати гарну модель, що перевикористовується.
- Легке написання модульних тестів. Оскільки бізнес-логіка реалізована у чистих функціях, протестувати її просто. UI залежить тільки від отриманих даних та займається виключно їх рендерингом. Так зручно тестувати UI через snapshot тести. Достатньо

налаштувати потрібний State і перевірити, що UI коректно рендерить його.

Але є й низка мінусів:

- Складнощі з модуляризацією. У нашій програмі вже були модулі. Вся бізнес-логіка була зібрана в модулі Core і кожній фічі потрібно імпортувати цей модуль собі (Рис. 2.8):

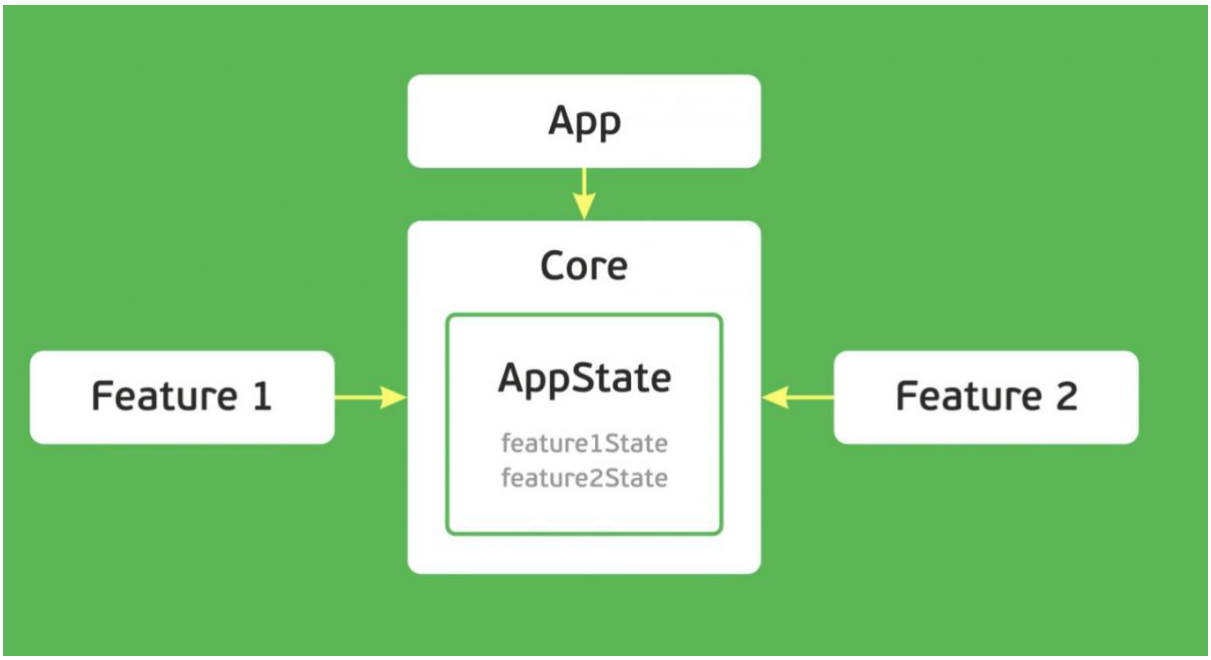


Рисунок 2.8 – Проблема модуляризації [8]

З одного боку, такий поділ дозволяв відокремити модель програми від UI. З іншого, модель вийшла монолітною та складною. Не було можливості відокремити частину логіки та використовувати окремо. Кожна фіча знала про модель всього додатка, а значить, і про інші фічі. З таким підходом подальше масштабування проекту лише погіршило б поточні проблеми [8].

- Проблеми із продуктивністю. Більшість UDF-фреймворків припускають наявність одного Store. Це дозволяє гарантувати єдине джерело правди та оновлювати State в одному місці. Але такий підхід веде до проблем із продуктивністю. Через те, що в Store приходять

Action з усієї програми, оновлення AppState можуть відбуватися дуже часто. Це створює велике навантаження як на Reducer, і на UI.

Покладаємо, є додаток і компонент у ньому. Наприклад, компонент для відображення даних користувача або інформації про замовлення. У цьому компоненті є стан, який лежить в AppState. Компонент знає про AppState, підписується на нього і при зміні відтворює дані на екрані (Рис. 2.9).

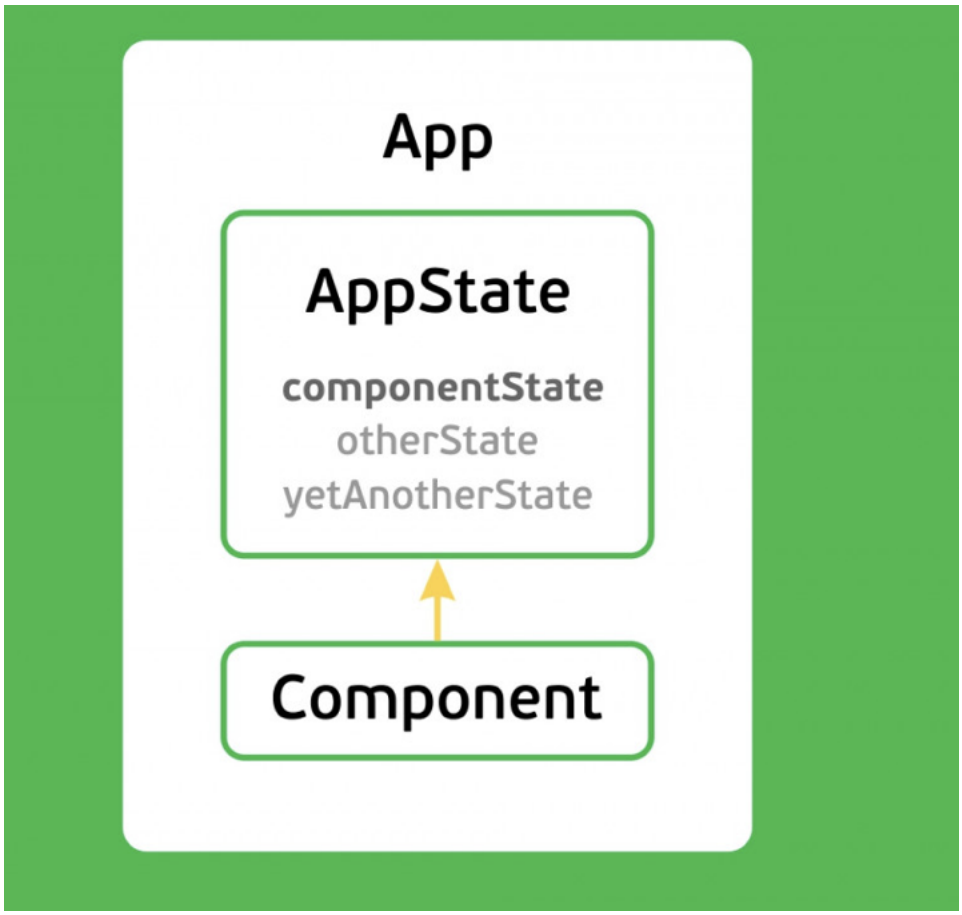


Рисунок 2.9 – Проблема вкладеності стейтів [7]

Таких компонентів у проекті може бути безліч, і кожен із них знає про AppState. Це погано, тому що такий підхід руйнує Закон Деметри. Більшість даних компонентів потрібні тільки власному державі, але їм належить пробиратися сквозь AppState до себе. Часто в проекті зазначаємо такий код (Рис 2.10).

```
let myCurrentState =
appState.innerState.anotherInnerState.andAnotherOneState.finallyNeededSt
ate
```

Рисунок 2.10 – Проблема вкладених станів

Писати і підтримувати такий код складно, хоча такий підхід і працює. Ситуація змінюється, коли ми захотіли перевикористовувати наш компонент в іншому прикладі. Для цього потрібно винести його в окремий модуль (Рис. 2.11).

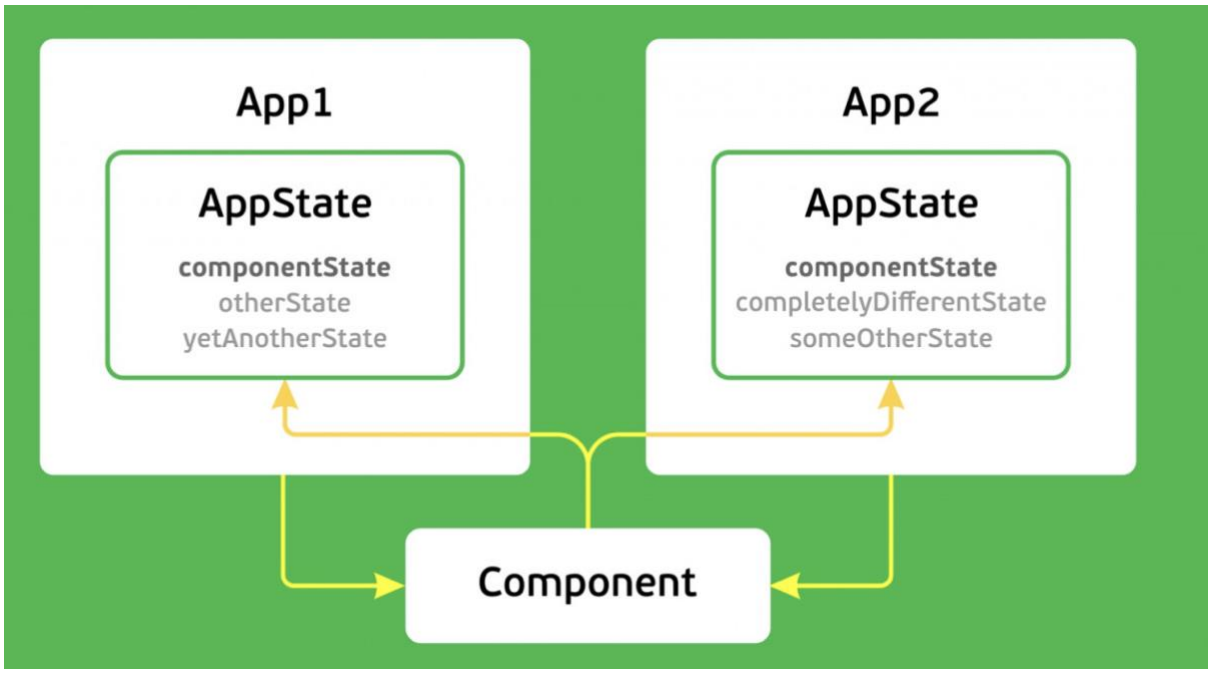


Рисунок 2.11 – Винесення компоненту в окремий модуль [8]

Получаємо відразу 2 проблеми:

App1 потрібно знати про компонент, щоб його використовувати. Компонент потрібно знати про App1, щоб отримати доступ до AppState. Ми отримуємо циклічну залежність між модулями, що не сподобається компоновщику Swift.

Компоненту тепер потрібно знати про App2, а в перспективі і про App3, App4 і так далі.

Вирішити проблему можна, забрав стан компонента всередині його модуля (Рис. 2.12).

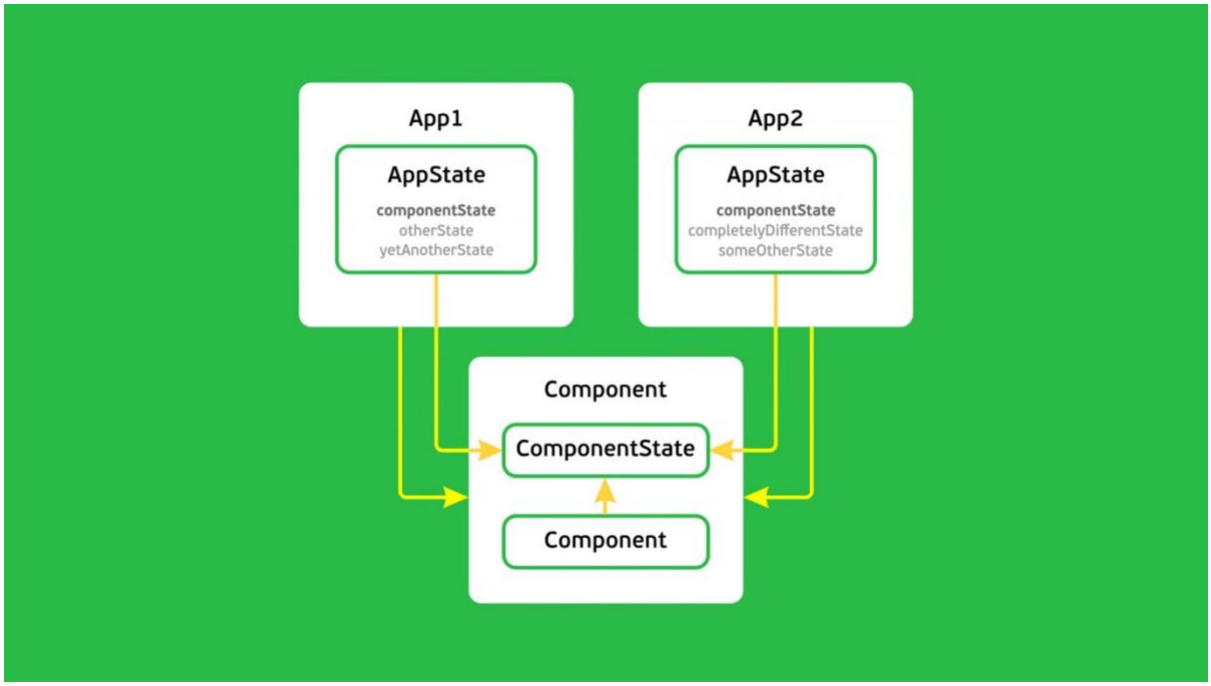


Рисунок 2.12 – Винесення стейту компоненту всередині модуля [8]

Тепер компоненти модуля ні про кого не знають і не мають ніяких залежностей. Сам компонент знає тільки про свій Стейт. Модулі застосунка знають про модулі компонентів, і AppState кожного з додатків використовує компоненти State зі свого модуля. Остається навчити компонент слухати тільки ту частину стейта, в якій він зацікавлений.

Інша проблема наступає коли два модуля хочуть використовувати одну й ту саму частину функціоналу. Припустимо, є дані профілю користувача, які ми хочемо використовувати у двох модулях. Обидва модулі використовуються в нашому додатку. Логічно, що стейт кожного з модулів повинен містити стейт профілю, а AppState містить дані і модулів, і профілю (Рис. 2.13).

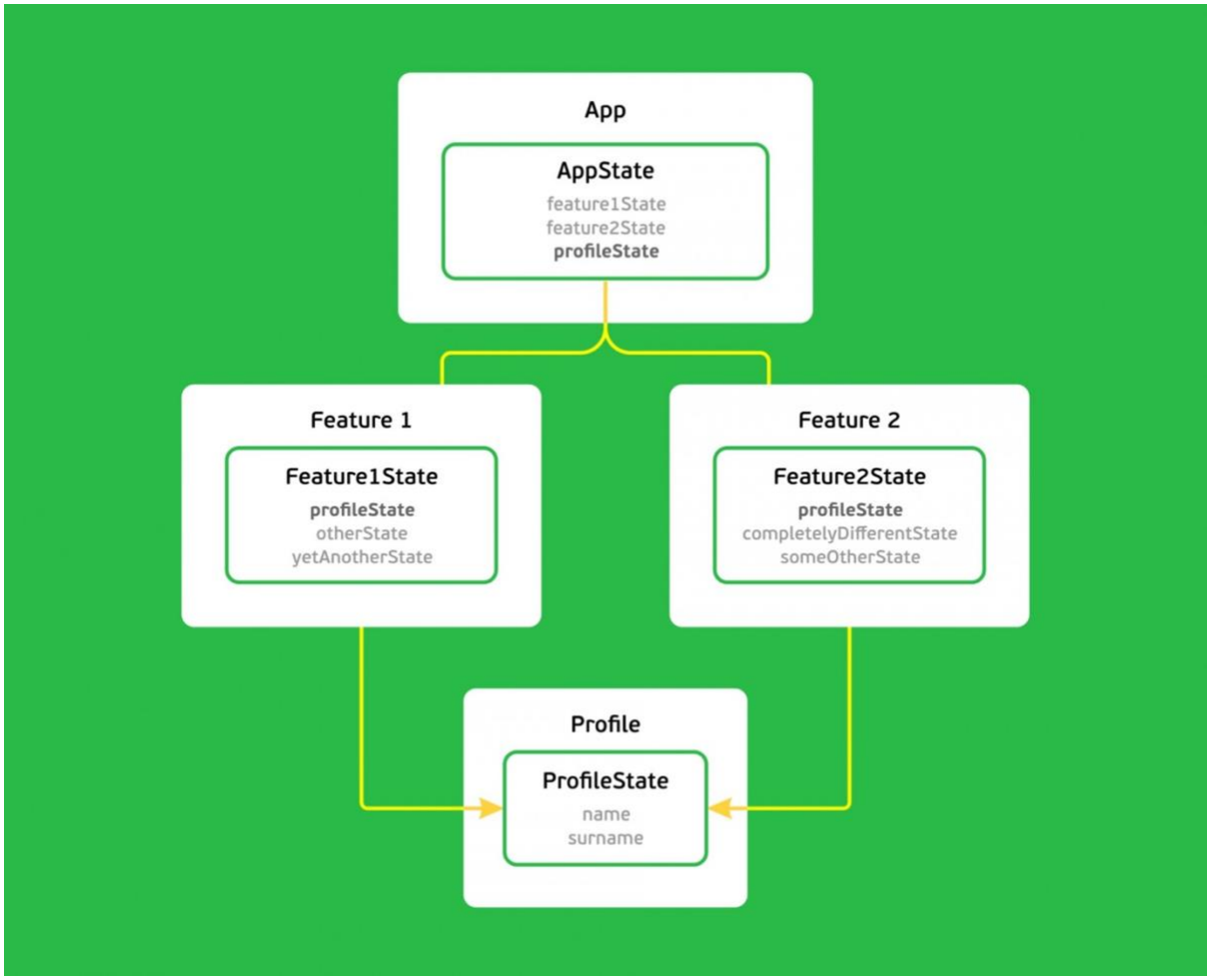


Рисунок 2.13 – Використання Profile у двох різних модулях [7]

У такому випадку отримуємо копії ProfileState у 3 місцях: AppState, Feature1State та Feature2State. Користувач у додатку один, тому і ProfileState нам потрібен у єдиному екземплярі. При цьому Feature1 і Feature2 безперечно повинні мати у своїх стейтах ProfileState, оскільки доступу до AppState вони не мають. Розглянемо 2 рішення, які умовно називаються **Computed Module State** та **State Protocol and Where Clause**:

Computed Module State. Розділимо FeatureState на 2 частини. Безпосередньо FeatureState — дані, що належать лише одній фічі. І FeatureModuleState - FeatureState + перевикористовувані стейти (Лістінг 2.1):

Лістінг 2.1 – Вирішення проблеми вкладених станів

```
struct FeatureModuleState {
    let feature: FeatureState
    let profile: ProfileState
}
```

Тоді в AppState зберігатимемо тільки FeatureState і ProfileState:

```
struct AppState {
    let feature1: Feature1State
    let feature2: Feature2State
    let profile: ProfileState
}
```

ModuleState реалізуємо як обчислювані властивості:

```
extension AppState {
    var feature1Module: Feature1ModuleState {
        .init(feature: feature1, profile: profile)
    }

 var feature2Module: Feature2ModuleState {
    .init(feature: feature2, profile: profile)
}
```

```
    }
```

В результаті отримаємо таку схему взаємодії (Рис. 2.14).

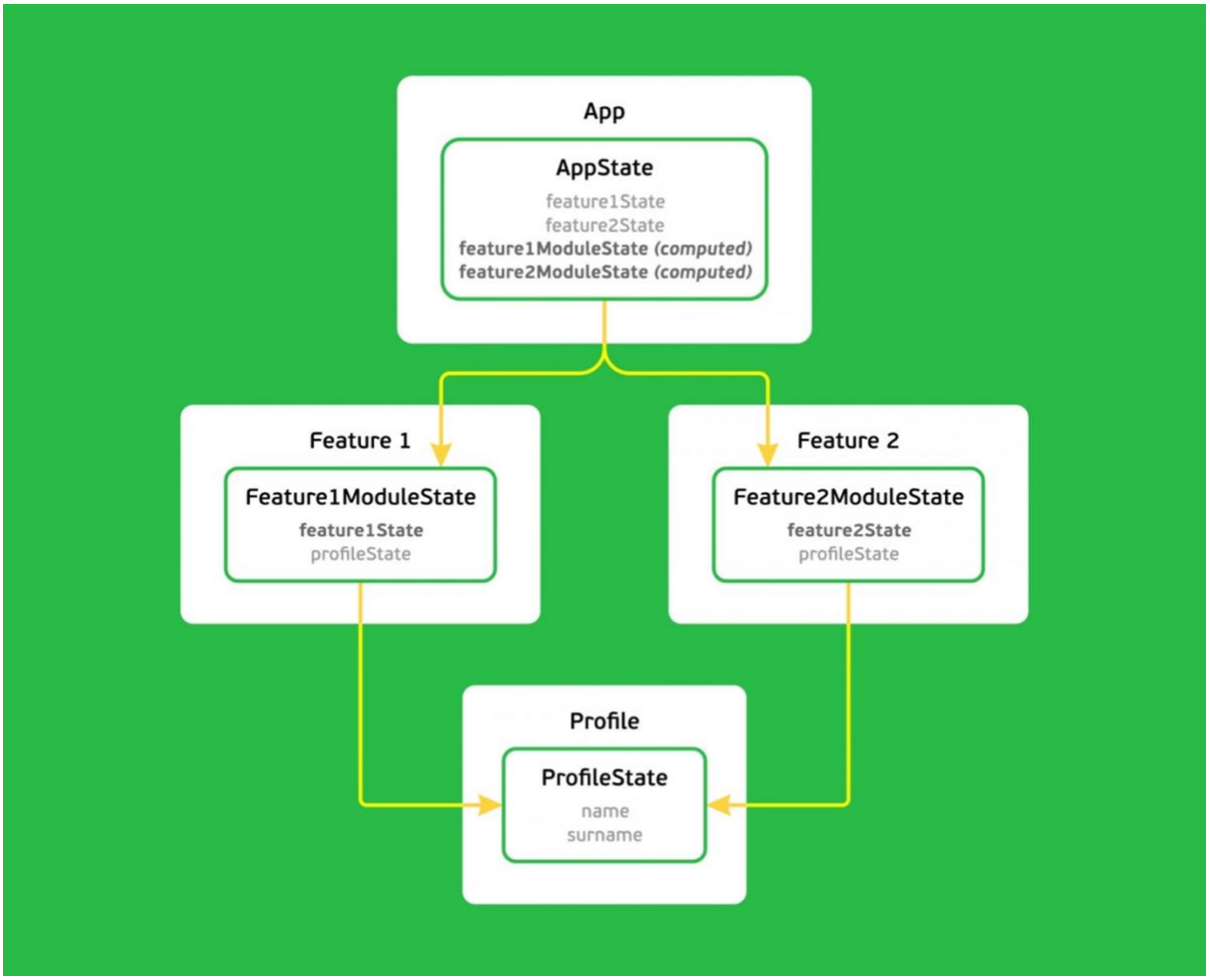


Рисунок 2.14 – Використання Profile у двох різних модулях [8]

На рівні AppState гарантується, що ProfileState у додатку завжди буде в єдиному екземплярі. ModuleState ж збираємо свій під кожен модуль. Такий підхід легко масштабується під будь-яку кількість фіч, що перевикористовуються, але вимагає реалізацію обчислюваної властивості під кожну нову фічу.

State Protocol and Where Clause. Замість обчислюваних властивостей використовуємо інтерфейси. Оголосимо інтерфейс Feature1ModuleState і реалізуємо його в AppState (Лістінг 2.2).

Лістінг 2.2 – State Protocol and Where Clause

```
protocol Feature1ModuleState {
    let feature: Feature1State
    let protifle: ProfileState
}
extension AppState: Feature1ModuleState {}
```

Аналогічно зробимо для Feature2. Тепер фічі можуть зберігати у себе FeatureModuleState і не знати, що AppState їх реалізує. Але є проблема - в Swift коваріантність працює тільки для системних дженериків, тому ми не зможемо Store<AppState> привести до Store<FeatureModuleState>.

У попередніх двох пунктах розглядався випадок, коли підписка на компонент відбувається в тому ж модулі, в якому знаходиться Store<AppState>, але це не завжди зручно. Якщо є модулі, в яких багато компонентів і які можуть динамічно підключатися і відключатися від Store, то було б зручно робити це безпосередньо в цих модулях. Але якщо передавати Store<AppState> в модуль повністю, ми отримаємо циклічну залежність. Головний модуль повинен знати про FeatureModule, щоб звернутися до нього. FeatureModule повинен знати про AppModule, тому що AppState лежить у AppModule (Рис. 2.15).

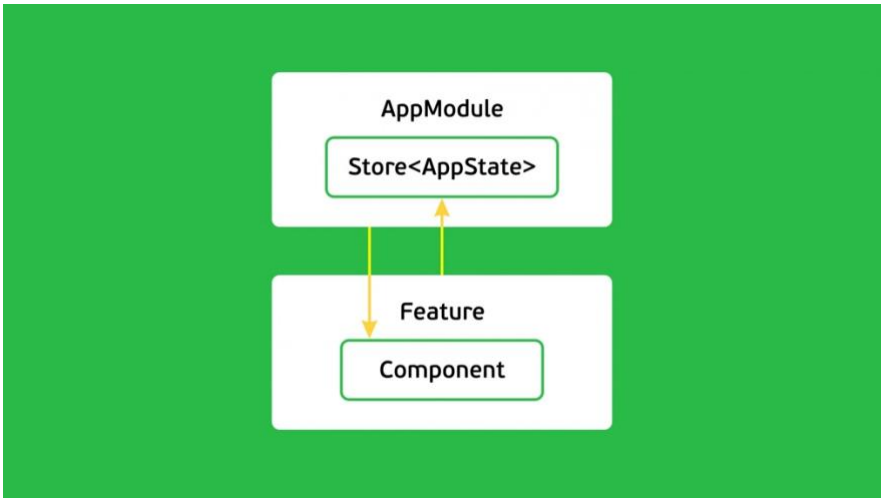


Рисунок 2.15 – Циклічна залежність [7]

Розглянемо повну схему вирішення цієї проблеми в UDF архітектурі (Рис 2.16).

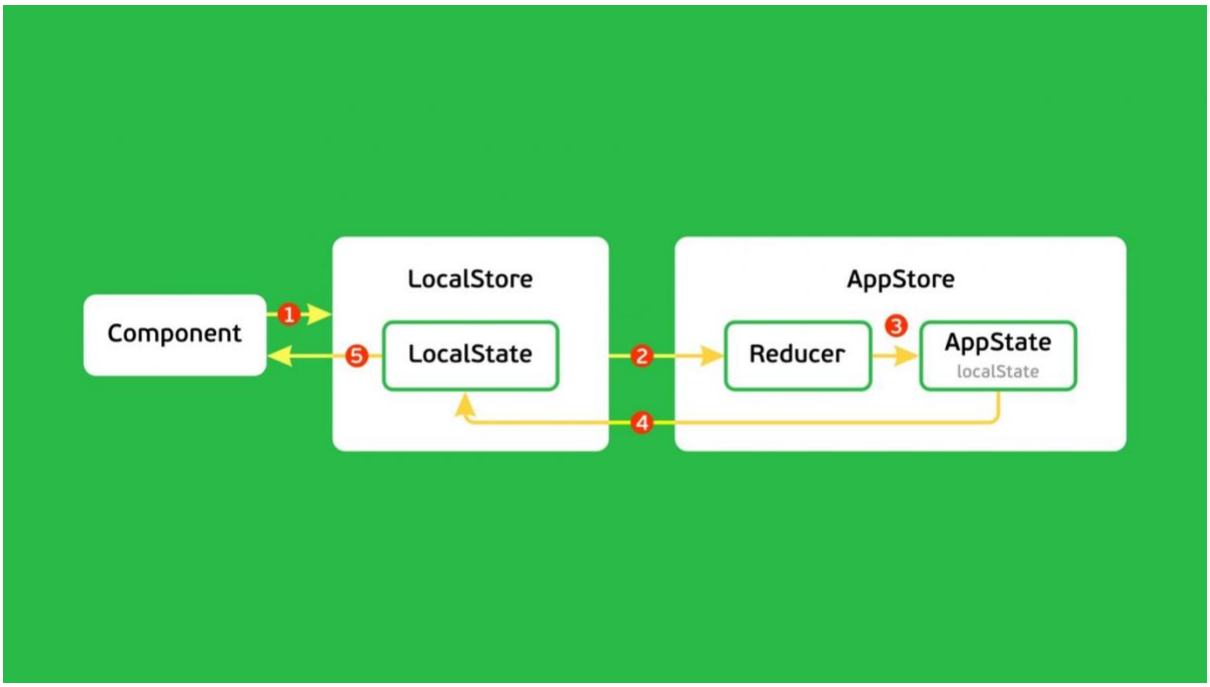


Рисунок 2.16 – Схема роботи архітектури з LocalStore [7]

1. Component відправляє Action в LocalStore.
2. LocalStore не обробляє Action, а транслює далі в батьківський Store.
3. Reducer батьківського Store обробляє Action та оновлює AppState.
4. LocalStore за підпискою отримує оновлений AppState і зберігає з нього LocalState.
5. Усі підписники LocalStore отримують оновлений LocalStore.

Також на 4 кроці є можливість порівняти новий та старий LocalState. Якщо він не змінився, ми можемо не повідомляти передплатників LocalStore про зміни. Це позбавляє цілі модулі реагування на кожен Action з інших модулів, якщо вони не впливають на їх власний стейт.

1.5 ВИСНОВКИ ДО РОЗДІЛУ

У MV(X) архітектурі через її імперативність не можливо сказати з високим ступенем упевненості, що враховуючи однакові дані з сервера, програма буде показувати те саме відображення. Це пов’язано з тим, що потенційно деякі частини відображення можуть бути оновленні без наявності контролю над ними, такі ситуацію зазвичай мають назву Сайд-ефекти. Вирішити ці проблеми у складних інтерфейсних програмах дозволяє концепція односпрямованого потоку даних.

Поширеною проблемою у застосуванні MV(X) є те, що неоптимальна початкова точка (незавершені шаблони, відсутність еталонних реалізацій) призвела до різноманітних втілень із іноді суперечливими, а іноді навіть неправильними інтерпретаціями. Тим часом Unidirectional Data Flow має більш задокументовані приклади використання, але з іншого боку, ця архітектура є більш складною і може мати проблеми с продуктивністю. З іншої сторони, MVVM має доволі однозначне розділення

2 ЗАГАЛЬНІ ПРИНЦИПИ ПРИ ПОБУДОВІ АРХІТЕКТУРИ МОБІЛЬНОГО ДОДАТКА

2.1 ЧИСТА АРХІТЕКТУРА ТА ЇЇ АДАПТАЦІЇ ПІД МОБІЛЬНІ ЗАСТОСУНКИ

Тим часом як VIPER і VIP привернули багато уваги на iOS. Android-CleanArchitecture, Wojda's Android Showcase3 є популярними прикладами для Android. Усі вони є додатками чистої архітектури Роберта Мартіна для мобільних додатків.

Clean Architecture [9] поєднує ідеї з попередніх архітектурних концепцій, таких як порти та адаптери (вона ж Hexagonal Architecture [10]) від Cockburn [11], Onion Architecture від Palermo [12], Elm архітектура [13] і додає принципи та концепції.

Clean Architecture пропонує використовувати чотири шари в стилі цибулі: об'єкти, варіанти використання, інтерфейсні адаптери, фреймворки та драйвери. (Рис. 3.1)

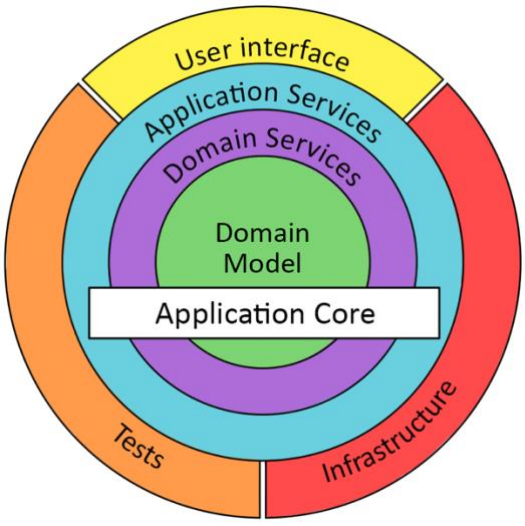


Рисунок 3.1 – Clean Architecture [10]

Його основними цілями згідно є відповідна незалежність від фреймворків і зовнішніх агентств (інтерфейс користувача, бази даних тощо) і можливість тестування. Як згадувалося вище, ці цілі корисні для

всіх видів програмного забезпечення, але вони особливо корисні для мобільних програм. Ключовою частиною досягнення цього є використання принципу інверсії залежностей для перетину меж кола. Класи внутрішнього циклу можуть спілкуватися з класами зовнішнього циклу лише через інтерфейси, реалізовані класами зовнішнього циклу (рис. 3.2).

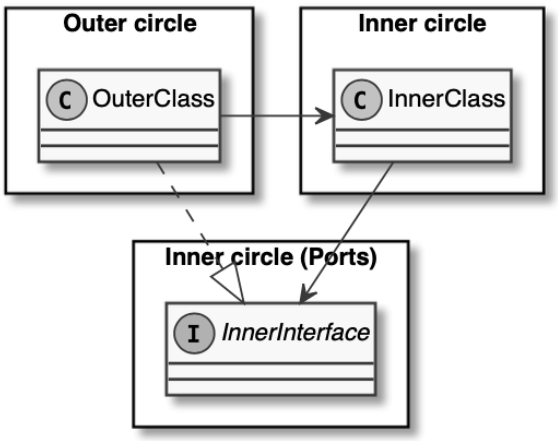


Рисунок 3.2 – Спілкування класів через інтерфейси [10]

Однією з головних проблем чистої архітектури є відсутність офіційних еталонних реалізацій для різних платформ (наприклад, у книгах Мартіна, його блозі, його навчальних відео та обліковому записі GitHub). Нижче наведена порівняльна таблиця щодо існуючих архітектур: різних варіацій Чистої архітектури. (Табл. 3.1)

Таблиця 3.1 надає оцінку чотирьох згаданих підходів з огляду на такі критерії:

1) Незалежність: Наскільки незалежним є підхід зовнішніх агентств/фреймворків?

2) Можливість тестування: наскільки прийнятним є підхід до модульних тестів, особливо локальних модульних тестів?

3) Модульність: чи підтримується та заохочується модульність?

4) Вірність: наскільки це відхиляється від ідей Чистої архітектури?

Таблиця 3.1 – Порівняльна характеристика існуючих реалізацій Чистої архітектури

	Незалежність	Можливість тестування	Модульність	Вірність
VIPER	Висока	Середня	Низька	Середня
VIP	Висока	Висока	Низька	Висока
Android-CA	Середня	Середня	Середня	Висока
Wojda	Висока	Висока	Висока	Середня

2.2 АНАЛІЗ ОСНОВНИХ ПРИНЦИПІВ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Патерни проектування

Говорячи про шаблони високого рівня та архітектурні концепції, ми не повинні забувати про шаблони проектування, такі як шаблони GoF [3], які, звичайно, також можна використовувати та інтегрувати незалежно.

Принципи SOLID

Чиста архітектура підтримує добре відомі принципи дизайну SOLID [9]. Оскільки вони сприяють досягненню загальних архітектурних цілей Clean Architecture, вони, природно, також актуальні для мобільних додатків.

Контроль інтерфейсу сутності

Для реалізації цього принципу було запропоновано структурувати системи з трьома типами об’єктів. Спочатку вибрані типи Entity-Interface-Control пізніше були змінені на Entity-Boundary-Control, а потім на Entity-Boundary-Interactor. Об’єкти сутності містять дані, які використовує система, і всю поведінку, природно пов’язану з цими даними. Граничні об'єкти моделюють інтерфейс із системою [8]. Уся поведінка, що залишилася, поміщається в об’єкти Interactor. Martin’s Clean Architecture адаптує ці типи об’єктів. [15] Граничні об’єкти, зокрема, сприяють роз’єднанню, а отже, незалежності та тестуванню. Однак EBI не дуже чітко пояснює, як отримати незалежність не тільки від інтерфейсу користувача. [16]

Впровадження залежностей

Впровадження залежностей допомагає підтримувати незалежність компонентів і їх тестування. Однак це не обов'язково означає, що потрібна структура впровадження залежностей. Може бути достатньо розглянути, які залежності мають бути гнучкими і організовувати їх у прозорий спосіб, який легко змінювати. [15]

Імперативна та декларативна парадигма

Також, розглянемо дві основні парадигми програмування: декларативну та імперативну.

Функціональне програмування є декларативною парадигмою програмування, на відміну від імперативних парадигм програмування.

Декларативне програмування — це парадигма, яка описує, «ЩО» робить програма, без явного визначення її потоку керування.

Імперативне програмування — це парадигма, яка описує, «ЯК» програма повинна щось робити, чітко вказуючи кожну інструкцію (або оператор) крок за кроком, що змінює стан програми [17].

Звичайно, зрештою, все компілюється в інструкції для ЦП. Отже, у певному сенсі декларативне програмування — це рівень абстракції поверх імперативного програмування.

У якийсь момент стан програми має бути змінено, щоб щось відбулося, і ці зміни можуть відбутися лише за допомогою інструкцій, що переміщують дані з одного місця (кеш-пам'яті, жорсткого диска...) до іншого.

Перетворення від декларативного до «імперативного коду», як правило, виконується механізмами, інтерпретаторами або компіляторами.

Наприклад, SQL є декларативною мовою. Використовуючи запит SELECT * FROM users WHERE id <= 100, висловлюється (або оголошуємо) бажання/намір: виділити перші 100 користувачів, коли-небудь зареєстрованих у базі даних. Спосіб отримання цих рядків

повністю делегований системі SQL: чи може вона використовувати індекс для прискорення запиту? Чи потрібно/може він використовувати кілька ядер ЦП, щоб закінчити раніше? [17]

Розробник, не має уявлення про те, як ці дані насправді отримуються. І розробника це не повинно хвилювати, якщо тільки він не досліджує будь-які проблеми продуктивності. Він вказує програмі, які дані хочемо отримати, а не як це зробити. Механізм/компілятор достатньо розумний, щоб знайти найоптимальніший спосіб зробити це [17].

Для мов, які використовують декларативну парадигму (наприклад, Haskell, SQL), цей «основний імперативний світ» абстрагований/прихований для розробників. Це те, про що розробники не повинні турбуватися.

Для мов із кількома парадигмами (наприклад, JavaScript, Scala) все ще існує можливість писати імперативний код. Це дозволяє користувачам писати декларативний код на основі імперативного коду, який вони написали самі. Це може бути корисно, наприклад, для підтримки функцій FP, які не вбудовані в мову, або просто для того, щоб зробити код більш «декларативним», що робить його більш читабельним і зрозумілим.

Імперативний код абстрагується декларативним, тобто тим, який використовують розробники для фактичного написання програмного забезпечення. Імперативна частина стає деталлю реалізації програмного забезпечення.

Тоді виникає питання, коли використовувати декларативну парадигму?

Цей питання стосується лише мов із кількома парадигмами. Очевидно, якщо розробник використовуєте функціональну мову, таку як Haskell, він завжди використовує декларативний код, але Swift це

мультипарадигмна мова програмування, в якій можливо використовувати як імперативний підхід, так і декларативний.

Отже, можна певною мірою зробити імперативний код схожим на декларативний. У такому випадку пропонується ізолювати імперативні частини від решти бази коду, щоб переконатися, що розробники використовують замість них «декларативні» функції. У мовах з кількома парадигмами шкала між декларативним і імперативним немає чіткого поділу на чорне і біле, а радше є кілька відтінків сірого. Від розробників залежить, який відтінок найкращий для конкретного проекту і команди. Тому було зроблено порівняльний аналіз обох парадигм (Табл. 3.2 і 3.3).

Таблиця 3.2 – Переваги та недоліки декладативної парадигми

Декларативна парадигма	
Переваги	Недоліки
Краща читабельність і розуміння коду	Більше рядків коду, де може ховатися потенційна помилка
Кращий контроль над фактичним виконанням змін у світі	Потенційна втрата продуктивності через збільшення обсягу пам’яті та виклики проміжних функцій
	Більш довга відладка програми завдяки більшим трасуванням стека
	Зазвичай розробникам такий спосіб програмування менш зручний

Таблиця 3.3 – Переваги та недоліки імперативної парадигми

Імперативна парадигма	
Переваги	Недоліки
Менше коду загалом, оскільки немає потреби загортати імперативний код у декларативні функції	Потрібно більше часу, щоб прочитати та зрозуміти, що робить код

Більш швидка відладка через менші трасування стека	Але загальне налагодження важче через мутації стану та «менш контрольовані» зміни світу
Зазвичай розробникам зручніше користуватися таким способом програмування	

2.3 ВИСНОВКИ ДО РОЗДІЛУ

В розділі проаналізовано ряд базових принципів проектування програмного забезпечення.

Поширеною проблемою чистої архітектури є доволі велика кількість різних реалізацій її принципів. Це призводить до того, що існують суперечливі, а іноді неправильні інтерпретації цих принципів.

Також було порівняно імперативний та декларативний підходи до написання коду. Вочевидь декларативний стиль має більше переваг. Найбільша перевага полягає в тому, що код стає більш читабельним і зрозумілим. Нерозуміння відповідальності певної частини бази коду є однією з найпоширеніших причин появи помилок. Це також одна з причин, чому додавання покращень і функцій займає більше часу, оскільки потрібно спочатку зрозуміти, що робить код, перш ніж вносити будь-які зміни. Функціональне програмування — це вираження того, «що» розробник хоче зробити з даними, але насправді нічого не робиться до останнього моменту. Щоб щось зробити, потрібно змінити стан і запустити оператори.

Ці частини обробляються механізмами/інтерпретаторами/ компіляторами, які знають, «як» ефективно робити «те, що» розробник написав в базі коду.

3. КОМБІНОВАНА АРХІТЕКТУРА ДЛЯ МОБІЛЬНИХ ЗАСТОСУНКІВ

3.1 ВИМОГИ ДО АРХІТЕКТУРИ ТА ПОСТАНОВКА ЗАДАЧІ ПРОЕКТУВАННЯ

Побудова мобільних додатків не завжди простий процес, оскільки мобільний пристрій має доволі багато апаратних та прикладних обмежень. Тому при побудові треба враховувати такі моменти:

- Велика кількість UI коду;
- Короткий реліз-цикл для софту та хардварі;
- Велика кількість різних сенсорів, датчиків у смартфоні;
- Збереження даних, їх синхронізація, а також підтримка режиму “Без інтернета”;
- Концепція програмування на життєвому циклі застосунка
- Код має бути ефективний, бо наявні обмеження по пам'яті та батареї смартфона;
- Багатопоточність з особливістю того, що оновлення відображення зазвичай трапляється в спеціальному потоку і тільки там;

Наданий список може бути не вичерпаним, оскільки при розробці будь-якого мобільного застосунку треба звертати увагу на вимоги бізнесу.

Наступним кроком слід зазначити, що будь-яка гарна та підтримуєма архітектура має на меті зберігати такі принципи: незалежність між своїми компонентами та фреймворками, а також можливість тестування.

Незалежність: незалежність між компонентами та фреймворками необхідна для побудови ефективної архітектури. Оскільки це може

допомогти зменшити прогалину між апаратними забезпеченням та виконанням програми, допомогаючи модульними тестами. Маючи більш гнучку архітектуру з'являється змога оновлювати застосунки за короткі проміжки циклу випуску, різноманітністю апаратного та програмного забезпечення та потребами налаштування.

Автоматизоване тестування: існує багато аспектів, які ускладнюють автоматизоване тестування для мобільних застосунків. В цей же час відомі переваги автоматизованого тестування. В розрізі мобільних застосунків ми можемо поділити автоматизоване тестування на чотири види: локальні модульні тести (запускаються на приладах для розробки), модульні тести на ізольованих машинах/смартфонах, UI тести, а також Snapshot тести. Серед трьох видів тестів слід відокремити модульні тести, бо вони надають нам швидких зворотній зв'язок, тому гарні архітектурні шаблони мають підтримувати модульні тести.

З огляду на існуючі архітектурні рішення пропонується створити комбіновану архітектуру: а саме поєднання класичних двоспрямованих архітектур MV(X) або Чистої архітектури з UDF (односпрямованою архітектурою). Тобто остаточний вибір MV(X) або різних реалізацій Чистої архітектури залежить від вимог зі сторони бізнесу, а також досвіду команди розробки.

Для цього пропонується створити DSL навколо імперативних підходів побудови компонентів відображення. В більшості випадках такий підхід виглядає більш привабливим в порівнянні з побудовою інтерфейсу у імперативному стилі або використанням XML конструкторів. Привабливість забезбечується через мінімізацію змінності даних, що покращує безпеку програми, оскільки незмінні структури даних у багатьох випадках менш схильні до помилок. Також пропонується побудова слою абстракцій, які будуть допомагати поєднувати UDF базовані архітектури з класичними MV(X). Це

необхідно для того, щоб в залежності від поставлених вимог створювати той, чи інший модуль в потрібній архітектурі. Також при побудові даної архітектури треба оптимізувати роботу UDF архітектур, де можуть виникнути проблеми з продуктивністю. Для цього пропонується використовувати композицію Стану з під-станів, а також використовувати можливість порівняння одноманітності об’єктів Станів, та оновлювати відображення в разі нерівності теперішнього і нового стану.

3.2 ПОБУДОВА І ПРИНЦИП РОБОТИ КОМБІНОВАНОЇ АРХІТЕКТУРИ

Очевидно, з першу необхідно визначити модулі загального призначення які не прив’язані до конкретної програми і можуть бути повторно використані будь-де. Такі модулі використовуються верхнім рівнем програми, і хоча вони також можуть мати залежність один від одного, але це рідкісний випадок. Тому цей рівень називається App Independent (Рис. 4.1).

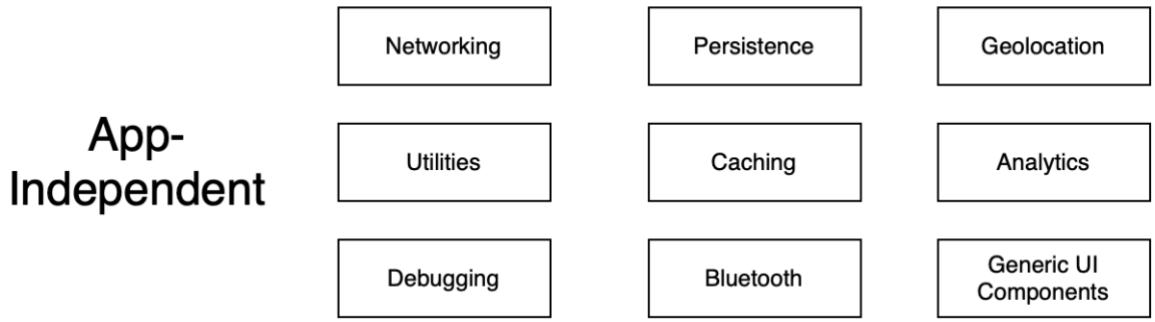


Рисунок 4.1 – App-Independent модуль

Наступним кроком при проектуванні архітектури є створення окремих модулів функціоналу застосунку. Зазвичай можна побачити наскільки тісно окремі модулі фунціоналу можуть залежати один від одного, що вказує на недоліки та слабкі сторони архітектури. Тому, основне правило, якого повинні дотримуватися розробники під час роботи над цими модулями — відсутність горизонтальної залежності на рівні функцій. Це означає, що один модуль функцій не повинен залежати від іншого. Архітектура кожного модуля може відрізнятися, тобто кожний модуль може бути реалізований ізольовано як і на UDF-подібній архітектурі, так і на MVX. (Рис. 4.2)

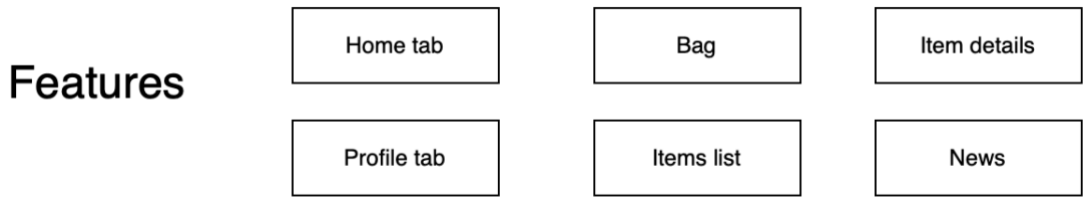


Рисунок 4.2 – Features модуль

Наступним кроком розробки архітектури за стосунку є створення компонентів, які з одного боку, належать до рівня Features, а з іншого — не належать до жодного конкретного модуля. Таким чином, пропонується створити ще один рівень, App Specific для застосунку (Рис. 4.3).

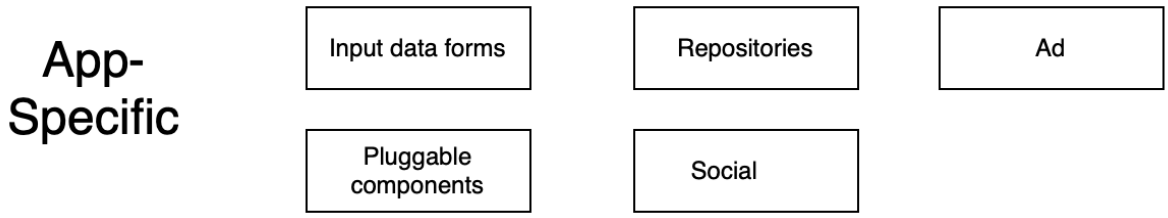


Рисунок 4.3 – App-Specific модуль

Host App – це верхній рівень архітектури. Цей рівень визначає стан програми та тип конфігурації. Він отримує сповіщення від ОС. (Рис. 4.4).

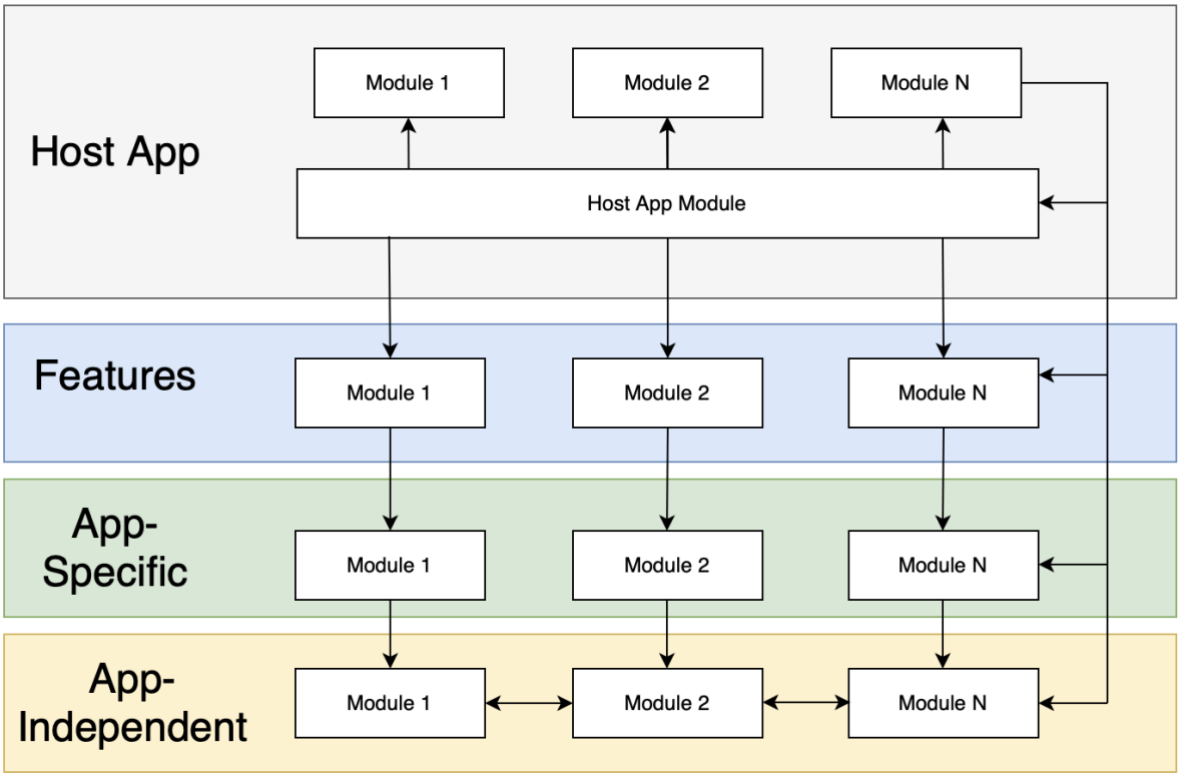


Рисунок 4.4 – Загальний вигляд запропонованої архітектури

Таким чином, Host App є єдиним рівнем, який відповідає за навігацію між функціями, конфігурацію функцій і зв’язок між ними. Щоб забезпечити низький зв'язок модулів, ми повинні дотримуватися принципу інверсії залежностей, щоб в рази підвищити придатність до відладки та тестування. Такі залежні модулі, як репозиторії, служби та аналітика, мають бути введені в модулі функцій за допомогою інтерфейсу з необхідними параметрами та налаштуваннями на рівні програми App Host.

Для здійснення переходів між функціями було запропоновано додати сутність Coordinator, яка належить кожному компоненту Features. Реалізація кожного Coordinator створюється в Host App модулі застосунку, який відповідає інтерфейсу навігації для певного модулю Feature. Цей підхід забезпечує детермінованість порядку показу екранів та логіку переходів. Приклад представлення збірки модуля «Список продуктів» і навігації від цієї функції до функції «Відомості про продукт»

за допомогою Coordinator (Рис. 4.5). Пунктирними лініями відображена композиція, а суцільними відображено взаємодію модулів між собою, тобто перехід з відображення «Список продуктів» до відображення «Відомості про продукт».

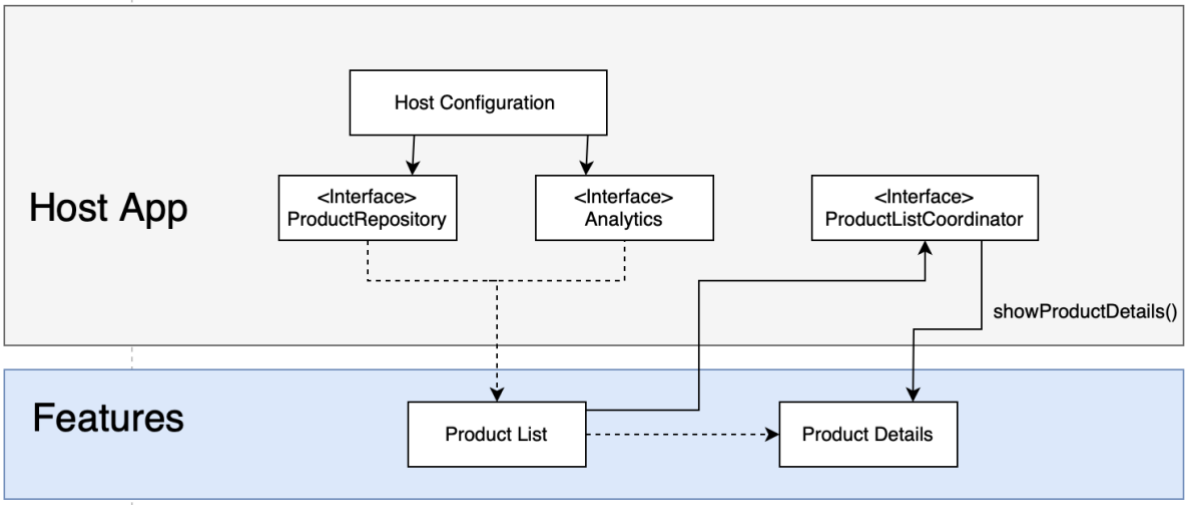


Рисунок 4.5 – Взаємодія Coordinator з модулями функціональності застосунку

Також у кожного модуля Feature буде існувати LocalState, цей локальний Стан модифікує виключно керуючий модуль (Reducer, Presenter, ViewModel) батьківського Store, після модифікації загального AppState, LocalStore за допомогою механізму підписки отримує оновлений LocalState та зберігає його. Наступним кроком, усі підписники LocalStore отримують оновлений LocalState.

3.3 СТВОРЕННЯ ДЕКЛАРАТИВНОГО ФРЕЙМВОРКУ ДЛЯ ПОБУДОВИ UI

У розробці iOS зазвичай існує два способи розробки інтерфейсу користувача. Перший спосіб - використовувати розкадровку або XIB. Другий спосіб - використовувати код програмно. Обидва методи вважаються менш ніж оптимальними для вимог побудови складних застосунків.

Storyboard або XIB підходить лише для малих і середніх проектів, оскільки їх відносно легко підтримувати. Але для великомасштабних проектів цьому методу не вистачає ефективності, оскільки його дуже важко підтримувати, і він особливо схильний до конфліктів, якщо два інженери працюють над однією розкадровкою або XIB. XIB також вважається неефективним, наприклад, якщо хочемо створити новий екран, нам потрібно буде створити файл .swift, а також файл .xib, отже, збільшивши розмір програми.

Побудова інтерфейсу користувача за допомогою програмного коду також має свої проблеми. За допомогою цього методу нам потрібно було б створити або запустити проект, щоб побачити результати, це займе надто багато часу та вплине на продуктивність інженерів. Створення складного інтерфейсу користувача потребує великої кількості коду, і потенційно буде дуже важко читати та налагоджувати код.

Сьогодні існує SwiftUI — рідна декларативна структура інтерфейсу користувача, розроблена Apple. Але, на жаль, SwiftUI можна використовувати лише на iOS 13 і вище, тим часом як у великої кількості компаній все ще є велика кількість користувачів з пристроями нижче iOS 13.

Ось чому було вирішено створити нову власну структуру інтерфейсу користувача. Нова структура інтерфейсу користувача

повинна підвищити продуктивність інженерів у створенні інтерфейсу користувача. Також існує прагнення гарантувати, що нова структура інтерфейсу користувача забезпечить найкращу продуктивність візуалізації, особливо зі складним макетом. Нарешті, також існує мета зробити структуру інтерфейсу користувача легкою для написання, більш передбачуваною в складній кодовій базі та легкою для рефакторингу.

Було обрано декларативну парадигму для нової структури інтерфейсу користувача замість імперативної, що означає, що маємо можливість просто вказати, що повинен робити інтерфейс користувача. SwiftyUIKit використовує декларативний стиль коду, щоб написаний код легше читати. Також хочеться переконатися, що за допомогою цієї декларативної парадигми стан інтерфейсу користувача буде більш передбачуваним, оскільки код є єдиним джерелом істини. SwiftyUIKit також має структуру та форму API, подібну до SwiftUI, щоб зробити його більш звичним і зручнішим у використанні для інженерів.

Для внутрішнього механізму інтерфейсу користувача було прийтяно рішення використовувати нативний фреймворк Apple UIKit. Це допоможе уникнути використання сторонніх бібліотек та бути залежними від них.

Декларативна програмна структура інтерфейсу користувача створюється також для швидкого циклу зворотного зв’язку для інженерів, щоб попередньо переглядати результати інтерфейсу користувача. Щоб досягти цієї мети, було створено функцію гарячого перезавантаження за допомогою сторонньої бібліотеки Injection [18]. Injection дозволяє оновити реалізацію коду інтерфейсу користувача в SwiftyUIKit у симуляторі iOS без необхідності перекомпонувати або перезапускати програму. Це економить розробнику значну кількість часу на коригування коду або повторення дизайну. Досвід розробки подібний до

hot-reload у Flutter. Він оновлює інтерфейс користувача в симуляторі щоразу, коли виявляє нові зміни.

SwiftyUIKit використовує ResultBuilder, це остання функція від Swift 5.6, яку можна використовувати для впровадження DSL для написання декларативного коду. (Рис. 4.6).

```
final class VStack: UIStackView {

    init(@StackBuilder views: () -> [UIView]) {
        super.init(frame: .zero)
        axis = .vertical
        translatesAutoresizingMaskIntoConstraints = false
        views().forEach { addArrangedSubview($0) }

    }

    required init(coder: NSCoder) {
        super.init(coder: coder)
    }
}

let vStack: VStack = VStack {
   UIView()
   UIButton()
   UIImageView()
}
```

Рисунок 4.6 – Приклад побудови вертикального стеку

Але, звісно, SwiftyUIKit набагато складніший, ніж приклад коду вище. Завдяки своїй архітектурі SwiftyUIKit створено як окрему структуру, тож якщо одного разу нам знадобиться змінити механізм

інтерфейсу користувача з UIKit на інший, не порушено поточну реалізацію на стороні клієнта. (Рис. 4.7)

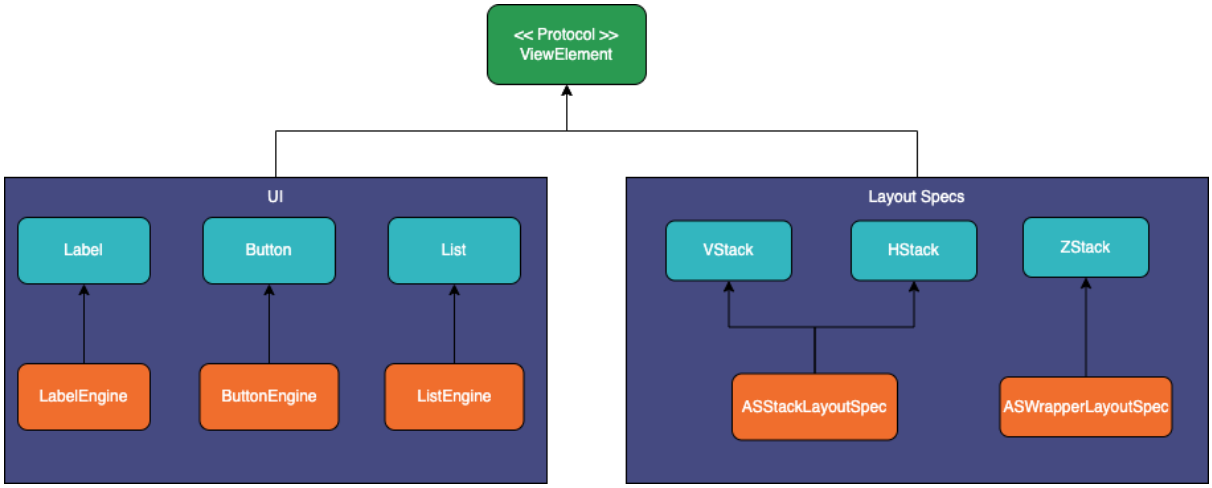


Рисунок 4.7 – Схема відокремлення реалізації фреймворку від внутрішнього рушія UIKit

Час, необхідний для написання моделей-обгорток у Swift вручну, був просто нерозумним, оскільки для створення фреймворку потрібні були не лише чисті моделі, але й спосіб конвертувати їх у обидві сторони.

Оскільки створення сотень моделей доволі рутиний процес, який займає доволі багато часу, тому було вирішено користуватися механізмами кодогенерування. Довелося згенерувати всі моделі та розширення для побудови фреймворку. Для цього було обрано існуючий фреймворк Sourcery [19].

Перш за все, було потрібно створити шаблон. Все, що знаходиться всередині {% %}, вважається потоком керування, все, що знаходиться всередині {{ }}, отримує доступ до значення та друкує його, а все інше просто друкується як відкритий текст [20]. Створений шаблон для кодогенерування виглядає так: (Рис. 4.8).

```
{% macro newlineIfNotLast isLast %}
{% if not isLast %}

{% endif %}
{% endmacro %}


{% for type in types.implementing.Chainable|class %}
// sourcery:file:{{ type.name }}+Chain.swift
import UIKit


public extension {{ type.name }} {
    {% for variable in type.allVariables|instance|openSet %}
    {% set isContains %}{% for parentVariable in type.inherits[type.inheritedTypes[0]].allVariables|instance|openSet %}{% if parentVariable.name == variable.name %}true{% endif %}{% endfor %}{% endset %}
    {% if not isContains == "true" %}
    {{variable.attributes.available}}
    @discardableResult
    func {{ variable.name }}(_ newValue: {{ variable.typeName }}) -> Self {
        {{ variable.name }} = newValue
        return self
    }
    {% call newlineIfNotLast forloop.last %}
    {% endif %}
    {% endfor %}
}
// sourcery:end
{% endfor %}
```

Рисунок 4.8 – Шаблон Sourcery для кодогенерування

Після цього утиліти Sourcery створюються усі необхідні розширення для моделей. (Рис. 4.9).

```
1  // Generated using Sourcery 1.0.0 — https://github.com/krzysztofzablocki/Sourcery
2  // DO NOT EDIT
3
4  import UIKit
5
6
7  public extension UILabel {
8      @discardableResult
9      func text(_ newValue: String?) -> Self {
10         text = newValue
11         return self
12     }
13
14     @discardableResult
15     func font(_ newValue: UIFont!) -> Self {
16         font = newValue
17         return self
18     }
19
20     @discardableResult
21     func textColor(_ newValue: UIColor!) -> Self {
22         textColor = newValue
23         return self
24     }
25
26     @discardableResult
27     func textAlignment(_ newValue: NSTextAlignment) -> Self {
28         textAlignment = newValue
29         return self
30     }
```

Рисунок 4.9 – Результати роботи Sourcery

3.4 ТЕСТУВАННЯ СТВОРЕНОГО ФРЕЙМВОРКУ

3.4.1 ТЕСТУВАННЯ ПЗ

Тестування програмного забезпечення (Software Testing) - перевірка відповідності реальних і очікуваних результатів поведінки програми, що проводиться на кінцевому наборі тестів, обраному певним чином [21].

Мета тестування - перевірка відповідності ПЗ вимогам, що пред'являються, забезпечення впевненості в якості ПО, пошук очевидних помилок в програмному забезпеченні, які повинні бути виявлені до того, як їх виявлять користувачі програми.

Для чого проводиться тестування ПО?

- Для перевірки відповідності вимогам.
- Для виявлення проблем на більш ранніх етапах розробки і запобігання підвищення вартості продукту.
- Виявлення варіантів використання, які не були передбачені при розробці. А також погляд на продукт з боку користувача.
- Підвищення лояльності до компанії і продукту, тому що будь-який виявлений дефект негативно впливає на довіру користувачів.

Забезпечення якості (QA - Quality Assurance) і контроль якості (QC - Quality Control) - ці терміни схожі на взаємозамінні, але різниця між забезпеченням якості і контролем якості все-таки є, хоч на практиці процеси і мають деяку схожість. QC (Quality Control) - Контроль якості продукту - аналіз результатів тестування і якості нових версій продукту, що випускається [21].

До завдань контролю якості відносяться:

- Перевірка готовності ПЗ до релізу;

- Перевірка відповідності вимог і якості даного проекту.

QA (Quality Assurance) - Забезпечення якості продукту - вивчення можливостей щодо зміни та поліпшення процесу розробки, поліпшення комунікацій в команді, де тестування є тільки одним з аспектів забезпечення якості.

До завдань забезпечення якості відносяться:

- Перевірка технічних характеристик і вимог до ПЗ;
- Оцінка ризиків;
- Планування завдань для поліпшення якості продукції;
- Підготовка документації, тестового оточення і даних;
- Тестування;
- Аналіз результатів тестування, а також складання звітів та інших документів.

Верифікація та валідація - два поняття тісно пов'язані з процесами тестування і забезпечення якості. На жаль, їх часто плутають, хоча відмінності між ними досить істотні.

Верифікація (verification) - це процес оцінки системи, щоб зрозуміти, чи задовольняють результати поточного етапу розробки умов, які були сформульовані в його початку.

Валідація (validation) - це визначення відповідності розробляється ПО очікуванням і потребам користувача, його вимогам до системи [21].

Також існують такі етапи тестування:

1. Аналіз продукту.
2. Робота з вимогами.
3. Розробка стратегії тестування і планування процедур контролю якості.
4. Створення тестової документації.
5. Тестування прототипу.

6. Основне тестування.
7. Стабілізація.
8. Експлуатація.

3.4.2 ОСНОВНІ ВИДИ ТЕСТУВАННЯ

Вид тестування - це сукупність активностей, спрямованих на тестування заданих характеристик системи або її частини, заснована на конкретних цілях [21].

Класифікація по запуску коду на виконання:

1. Статична тестування - процес тестування, який проводиться для верифікації практично будь-якого артефакту розробки: програмного коду компонент, вимог, системних специфікацій, функціональних специфікацій, документів проектування та архітектури програмних систем і їх компонентів.
2. Динамічне тестування - тестування проводиться на працюючій системі. Такий вид тестування не можливо виконувати без запуску вихідного коду програми.

Статичне тестування буде проводитися на етапі розробки застосунку за допомогою код рев'ю та статичних аналізаторів Swift коду. На даному етапі можуть бути усунуті незначні баги та помилки в виконанні програми, які можна помітити у коді програми.

На етапі динамічного тестування буде перевірятися коректна робота фреймворку на вже на iPhone та iPhone Simulator. Для цього будуть використовуватися модульні та інтеграційні тести. Нижче наведені приклади модульного і інтеграційного тестування (рис. 4.10 та 4.11).

```
func test_stackView_elements() {
    let view1 = UIView()
    let view2 = UIView()
    let view3 = UIView()

    let stackView = HorizontalStack {
        view1
        UIStackViewSpace(1)
        view2
        view3
    }

    XCTAssertEqual(stackView.arrangedSubviews, [view1, view2, view3])
    XCTAssertEqual(stackView.customSpacing(after: view1), 1)
    XCTAssertEqual(stackView.customSpacing(after: view2), UIStackView.spacingUseDefault)
}
```

Рисунок 4.10 – Приклад модульного тестування фреймворку

```
func test_stackview_dynamic_spacing() {
    let spaceProvider = UIStackViewDynamicSpaceProvider(1)
    let afterSpacedView = UIView()

    let stackView = HorizontalStack {
        afterSpacedView

        UIStackViewSpace(spaceProvider)
    }


    var currentSpace: CGFloat {
        stackView.customSpacing(after: afterSpacedView)
    }

    XCTAssertEqual(currentSpace, 1)
    XCTAssertEqual(currentSpace, stackView.customSpacing(after: afterSpacedView))

    spaceProvider.update(with: 5)
    XCTAssertEqual(currentSpace, 5)
    XCTAssertEqual(currentSpace, stackView.customSpacing(after: afterSpacedView))
}
```

Рисунок 4.11 – Приклад інтеграційного тестування фреймворку

3.5 ВИСНОВКИ ДО РОЗДІЛУ

З огляду на проведений аналіз, було запропоновано створити комбіновану архітектуру. Її особливість полягає в тому, що вона слідує основним принципам побудови і проектування програмного забезпечення. Було враховано попередній досвід чистої архітектури та надано більш детальний опис шарів комбінованої архітектури. Було враховано недоліки UDF побідних архітектур та запропоновано варіанти поліпшення або уникання цих проблем. Наприклад, декомпозиція єдиного глобального Стану на під-Стани, а також їх порівняння для того, щоб уникнути випадків оновлення відображення користувача, коли це не потрібно. Також, було запропоновано з огляду на переваги декларативного підходу до створення коду, створити декларативний DSL-фреймворк для побудови інтерфейсу користувача. DSL-фреймворк був протестован як модульними так і інтеграційними тестами. Під час побудови цього фреймворку було використано останні можливості мови програмування Swift. Створений фреймворк надає розробникам можливість більш швидкої та гнучкої побудови модулів графічних зображень.

4. РОЗРОБКА СТАРТАП-ПРОЕКТУ «РОЗРОБКА КОМБІНОВАНОЇ АРХІТЕКТУРИ МОБІЛЬНИХ ДОДАТКІВ НА ОСНОВІ АНАЛІЗУ АРХІТЕКТУР MV(X) ТА UNIDIRECTIONAL DATA FLOW»

4.1 ОПИС ІДЕЇ ПРОЕКТУ

Описана технологія може бути реалізована в якості мобільного застосунку. Цей розділ ставить перед собою мету реалізації технології використання побудови архітектурних моделей для управління магазином: визначення необхідного рівня запасів на складах та ціноутворення, а також проведення маркетингового аналізу та виявлення ринкових можливостей використання роботи. Процес включає в себе:

- імплементацію веб-додатку з використанням підходів та технологій описаних у попередніх розділах роботи;
- розробку стратегії виходу конкурентоспроможного продукту на ринок та подальший розвиток стартапу.

Для отримання цілісного уявлення про зміст ідеї та можливі базові потенційні ринки, в межах яких потрібно шукати групи потенційних клієнтів важливо побудувати таблицю з описом ідеї можливими напрямами застосування та основними вигодами, що може отримати користувач товару.

Таблиця 4.1 – Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Створення DI веб-додатку для для управління магазином	1. Визначення необхідного та оптимального рівня запасів на складах	Користувач буде мати змогу застосовувати додаток , щоб оптимізувати рівні запасів на основі моделей попиту, гарантуючи, що вони задовольняють запити споживачів, оптимізуючи прибуток
	2. Рекомендації щодо ціноутворення	Користувач буде мати змогу застосовувати додаток , щоб оптимізувати стратегії ціноутворення

Отже, ідея проекту заключається в тому, що через веб-додаток менеджери магазинів матимуть змогу отримати рекомендації щодо об'ємів закупок товарів для подальшої реалізації та щодо ціноутворення. Також додаток матимо аналітику, яка показуватиме результативність обраних дій.

Визначений перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару є підґрунтям для формування його конкурентоспроможності. Тому у таблиці 8.2 показано чим товар відрізняється від існуючих аналогів чи замінників на основі техніко-економічних властивостей та характеристик.

Таблиця 4.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики і ідеї	(потенційні) товари/концепції конкурентів				W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Мій проект	Конкурент 1	Конкурент 2	Конкурент 3			
1.	Форма виконання	Мобільний-додаток	Десктопний додаток	Десктопний додаток	Веб-додаток			+
2.	Собівартість	Низька	Низька	Низька	Висока		+	
3.	Емоційність	Контрольована	Випадкова	Випадкова	Випадкова			+
4.	Крос-платформенність	Так	Ні	Ні	Так			+
5.	Потреба в інтернеті	Так	Так	Ні	Так		+	

Сильними сторонами даного проекту є те, що форма виконання ідеї – мобільний додаток, що є більш гнучким способом взаємодії з програмою, оскільки не вимагає використання комп’ютера, лише достатньо телефоону, а також можлива крос-платформенність, адже здатність підтримувати велику кількість платформ напряму збільшує кількість потенційних клієнтів. Всі інші характеристики є нейтральними.

Тому даний проект можна вважати конкурентоспроможним.

4.2 ТЕХНОЛОГІЧНИЙ АУДИТ

В межах даного підрозділу необхідно провести аудит технології, за допомогою якої можна реалізувати ідею проекту (технології створення товару).

Таблиця 4.3 – Технологічна здійсненність ідеї проекту

п/п	Ідея проекту	Технології її реалізації	Наявність технологій	Доступність технологій
1.	Використання засобів Комбінованої архітектури для створення магазина	Комбінована архітектура	Наявна	Безкоштовна, доступна
		MV(X) архітектурні моделі	Наявна	Безкоштовна, доступна
		UDF архітектурні моделі	Наявна	Безкоштовна, доступна
Обрані технологііі реалізації проекту для створення мобільного додатку за допомогою використання комбінованої архітектури, ця архітектура є безкоштовною та зручною у використанні, адже поєднує переваги MV(X) та UDF архітектурних моделей.				

Отже, проект буде реалізовано за допомогою комбінованої архітектури, адже надає можливість поєднувати переваги інших архітектурних моделей. Тому даний проект можна вважати технологічно здійсненним.

4.3 АНАЛІЗ РИНКОВИХ МОЖЛИВОСТЕЙ

Визначення ринкових можливостей, які можна використати під час ринкового впровадження проекту, та ринкових загроз, які можуть перешкодити реалізації проекту, дозволяє спланувати напрями розвитку проекту із урахуванням стану ринкового середовища, потреб потенційних клієнтів та пропозицій проектів- конкурентів.

Спочатку проводимо аналіз попиту: наявність попиту, обсяг, динаміка розвитку ринку.

Таблиця 4.4 – Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців, од	100
2	Загальний обсяг продаж, грн/ум.од	1000
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу	Висока вартість початкового капіталу
5	Специфічні вимоги до стандартизації та сертифікації	-
6	Середня норма рентабельності в галузі (або по ринку), %	R = (3000000 * 100) / (1000000 * 12) = 25%

Отже, середня норма рентабельності в галузі менша, ніж банківський відсоток на вкладення. Тому має сенс вкласти кошти в саме цей проект, адже проект не має специфічних вимог до стандартизації та сертифікації, бо він буде виконаний у вигляді веб-додатку.

Далі визначаються потенційні групи клієнтів, їх характеристики, та формується орієнтовний перелік вимог до товару для кожної групи (табл. 4.5).

Таблиця 4.5 – Характеристика потенційних клієнтів стартап-проекту

п/п	Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
	Оптимізація роботи магазинів	а) Магазини, яким потрібно оптимізувати об'єми товарів на складах; б) магазини, яким потрібно оптимізувати стратегії ціноутворення	б) націлені на постійне поповнення різноманіття послуг, середнє спілкування з клієнтом б) націлені на постійне поповнення різноманіття послуг, середнє спілкування з клієнтом	Мобільний-додаток: Надійний, Функціонально розвинений, зручний дизайн; Постачальник веб-сервісу: Надає підтримку, Постійно покращує рішення

Визначено характеристики стартап-проекту: основну потребу, що формує ринок - оптимізацію роботи магазинів; наведено основні цільові сегменти ринку – компанії з великими відділами підтримки, продажів, замовлень тощо; відмінності у поведінці різних потенційних цільових груп клієнтів - отримання інформації та різноманітне спілкування; затверджено основні вимоги до споживачів

– надійні та високофункціональні чат-боти з яскраво вираженою емоційною складовою.

Далі складаються таблиці факторів, що сприяють ринковому впровадженню проекту, та факторів, що йому перешкоджають (табл. 4.6-4.7).

Таблиця 4.6 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Зростаюча вимогливість покупців	Споживачі вимагають більш точних прогнозів та рекомендацій від алгоритмів	Створення дослідницької лабораторії, присвяченої покращенню роботи алгоритмів
2	Зміна потреб і смаків покупців	Окрім служб підтрики/продажу клієнти можуть придумати інші застосування, потрібні на тому чи іншому етапі розвитку клієнтського продукту	Додавати нові проекти та функції до списку послуг

3.	Збільшення витрат на технічну підтримку	Невчасне реагування на сучасний ринок потреб користувачів	Вчасно оновлювати програмне забезпечення
4.	Конкуренція	Вихід на ринок великої компанії	Запропонувати великій компанії поглинути себе; передбачити додаткові переваги власного сервісу для того, щоб повідомити про них саме після виходу міжнародної компанії на ринок
5.	Зменшення кількості замовників	Зменшення зацікавленості замовників у наданих сервісах	Постійні оновлення алгоритму

Було наведено основні фактори загроз стартап-проекту. Найбільшою загрозою для проекту є зміна потреб користувачів (користувачам потрібен новий функціонал) та їх вимогливість (користувачам потрібні більш точні прогнози та рекомедації від алгоритмів). Для зменшення цих загроз потрібне створення дослідницької лабораторії, присвяченої покращенню роботи алгоритмів, а також потрібне своєчасного оновлення програмного забезпечення і користувацького інтерфейсу. Найменшими загрозами є збільшення витрат на технічну підтримку та зменшення кількості користувачів замовників.

Таблиця 4.7 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Підвищились технологічні бар’єри входу на ринок: розробка з використання комбінованої архітектури – не тривіальна задача	Для реалізації функціональності, задовільної для користувачів, потрібно наймати команду з досвідом побудови складних архітектурних рішень	Наймати та навчати кадри, які могли б революціонувати область
2	Можливість швидкого розвитку у зв'язку з різким зростанням попиту на ринку	Майже всі власники магазинів хотіли б оптимізувати роботу закладів та збільшити прибутки, тому користувачів багато	Максимальне захоплення ринку за рахунок впливових клієнтів
3	Зростання можливостей потенційних покупців	Зростання фінансування досліджень у галузі машинного навчання	Запропонувати свої послуги зацікавленим підприємствам
4	Зниження довіри до конкурента 1	У додатку конкурента 1 нещодавно було знайдено витік інформації, яка збиралася для аналітики	При виході на ринок звертати увагу покупців на безпеку нашого додатку

5	Зменшення витрат на технічну підтримку	Збільшення продуктивності роботи штату компанії за рахунок підвищення їхнього професійного рівня	Підвищувати рівень кваліфікації своїх співробітників

Було наведено основні фактори сприяння ринковому впровадженню проекту: підвищення технологічних бар'єрів входу на ринок: розробка додатків за допомогою комбінованої архітектури – не тривіальна архітектурна модель, але яка має можливості швидкого розвитку у зв'язку з різними зростаннями попиту на ринку та зростання можливостей потенційних покупців. Основними реакціями компанії є: надання своїх послуг зацікавленим підприємствам, безпечність програмного забезпечення, найм та навчання кадрів, які могли б працювати з комплексною архітектурою область та максимальне захоплення ринку за рахунок впливових клієнтів

Надалі визначаються загальні риси конкуренції на ринку (табл. 4.8).

Таблиця 4.8 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - олігополія	Існує мала кількість фірм-конкурентів на ринку	Створення більш технологічно досконалих рішень, ніж конкуренти
2. За рівнем конкурентної - інтернаціональна	Майже всі конкуренти - закордонні	Можна знайти дешевших дослідників

3. За галузевою ознакою - внутрішньогалузева	Конкуренти мають сервіси, які використовуються лише всередині даної галузі	Добре описані рамки галузі та способи впливу на неї
4. Конкуренція за видами товарів: - товарно-видова	Види товарів є однаковими, а саме - програмне забезпечення	Конкуренція між різними архітектурними шаблонами
5. За характером конкурентних переваг - нецінова	Вдосконалення технології створення дотатку, щоб собівартість була нижчою	Поліпшення якості продукції
6. За інтенсивністю - не марочна	Бренди відсутні	Легше вийти на ринок молодій компанії

Було наведено проведено аналіз конкуренції на ринку, а саме визначено: тип конкуренції - олігополія; конкуренція за рівнем конкурентної боротьби - міжнародна; конкуренція за галузевою ознакою - внутрішньогалузева; конкуренція за видами товарів - товарно-видова; конкуренція а характером конкурентних переваг - нецінова; конкуренція за інтенсивністю - не марочна. Також було наведено можливі дії компанії, щоб бути конкурентоспроможною: створення більш технологічно досконалих рішень, ніж конкуренти; пошук дешевших дослідників; використання способів впливу на галузь; поліпшення якості продукції.

Далі розробляється перелік факторів конкурентоспроможності для ринку на основі аналізу складових моделі 5 сил М. Портера (табл. 4.9).

Таблиця 4.9 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	Google	Бар’єр - потужний дослідницький відділ	Amazon Google Сильні сторони: стабільні, мають підтримку	Rozetka eBay	-
Висновки:	Дуже інтенсивна боротьба за першість в архітектурних моделях для мобільних застосунків	Потенційно може вийти будь-яка компанія з необхідними ресурсами	Нічого не диктують, в крайньому разі можна і самому серверів накупити	Диктують умови	-

Отже, з огляду на конкуренту ситуацію можна з впевненістю сказати, що проект має можливість роботи на ринку, тому що серед наведених конкурентів немає тих, які б могли його потіснити, адже розроблене рішення спрощує та пришвидшує роботу спеціаліста.

На основі аналізу висновків, наведених вище, визначається та обґрунтовується перелік факторів конкурентоспроможності (табл. 4.10).

Таблиця 4.10 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, що роблять фактор для порівняння конкурентних проектів значущим)
1	Науково-технічний потенціал	Кращі дослідження=вища якість послуг
2	Кадровий потенціал	Кращі кадри=кращий рівень розробки

Було наведено основні фактори конкурентоспроможності, які будуть представлені на ринку, а саме: науково-технічний потенціал, який дозволяє надавати вищу якість послуг, а також кадровий потенціал, який гарантує більш швидкі та більш якісні дослідження у галузі DI.

За визначеними факторами конкурентоспроможності проводиться аналіз сильних та слабких сторін стартап-проекту (табл. 4.11).

Таблиця 4.11 – Порівняльний аналіз сильних та слабких сторін «Емоційна діалогова система для рекомендації онлайн курсів»

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні зі стартап-проектом						
			-3	-2	-1	0	1	2	3
1	Науково-технічний потенціал	19			+				
2	Кадровий потенціал	19			+				

Було наведено порівняльний аналіз сильних сторін проекту товарів- конкурентів і нашого підприємства. Найбільше балів набрано для таких факторів конкурентоспроможності – науково-технічний потенціал та кадровий потенціал.

Далі проводимо SWOT-аналіз (табл. 4.12).

Сильні сторони: найкращі фахівці в області	Слабкі сторони: слабкий маркетинговий відділ, відсутність репутації, молода компанія
Можливості: стрімкий ріст ринку та технологічності рішень у сфері архітектурних шаблонів	Загрози: конкуренти значно більш відомі і потужніші, мають більше ресурсів, з часом здатні придумати кращу модель або покращити нашу

Отже, внутрішні можливості компанії і спроможності щодо виведення продукту на ринок характеризуються такими сильними і слабкими сторонами: сильні - найкращі фахівці в області; слабкі - слабкий маркетинговий відділ, відсутність репутації, молода компанія. Ринкові та можливості компанії щодо зовнішнього оточення характеризуються можливостями і загрозами: можливості - стрімкий ріст ринку та технологічності рішень; загрози - конкуренти значно більш відомі і потужніші, мають більше ресурсів, з часом здатні придумати краще поліпшення моделі або покращити нашу.

На основі SWOT-аналізу складаються альтернативи ринкового впровадження стартап-проекту (табл. 4.13).

Таблиця 4.13 – Альтернативи ринкового впровадження стартап-проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Покращуємо алгоритм настільки, щоб конкурентів залишити далеко позаду	50%	2 роки

2	Виходимо на ринок з обмеженим функціоналом, граємо на швидкому рості ринку	50%	1 рік

Отже, було визначено альтернативу ринкового впровадження стартап-проекту - вихід на ринок з обмеженим функціоналом. Оскільки час на покращення алгоритмів є значно більшим та не пропонує більшу ймовірність отримання ресурсів.

4.4 РОЗРОБКА РИНКОВОЇ СТРАТЕГІЇ ПРОЕКТУ

Розроблення ринковоїстратегії першим кроком передбачає визначення стратегії охоплення ринку: опис цільових груп потенційних споживачів (табл. 4.14).

Таблиця 4.14 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних Клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Магазини, яким потрібно оптимізувати роботу складів	Готові, якнайшвидше	Попит - високий	Не дуже інтенсивна	Середня
2	Магазини, яким потрібно оптимізувати стратегії ціноутворення	Готові, якнайшвидше	Попит - середній, через недовіру	Не дуже інтенсивна	Середня

			технологія м		
Які цільові групи обрано: обрано обидві групи, так як вони майже не відрізняються, а проект їх може задовольнити.					

Отже, було вибрано основні цільові групи: компанії, які потребують автоматизацію відділу підтримки та компанії, які потребують автоматизацію відділу продажів чи прийняття замовлень, адже обидві групи мають схожі вимоги та готові сприйняти продукт. Далі визначається базова стратегія розвитку (табл. 4.15).

Таблиця 4.15 – Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку*
1	Створення застосунку за допомогою комбінованої архітектури що є новою і надійною властивістю	Створення продуктових інновацій та їх маркетинг	Основна позиція - новизна, адже є ключовою технологією і перевагою нашого проекту	Стратегія диференціації

Наступним кроком є вибір стратегії конкурентної поведінки (табл. 4.16).

Таблиця 4.16 – Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
1	Ні	Шукати нових і забирати існуючих	Стабільність/надійн ість сервісів, підтримка, інтерфейс	Оборонна

Отже, було визначено базову стратегію конкурентної поведінки - оборонну, також матиме технічну підтримку та зручний інтерфейс, що формуватиме довіру і прихильність споживачів.

Далі визначається стратегія позиціонування проекту, яка допоможе користувачам ідентифікувати програмний продукт (табл. 4.17).

Таблиця 4.17 – Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
1	Коректна робота, надійність, підтримка	Стратегія диференціації	Основна позиція - емоційність, адже є ключовою технологією і перевагою нашого проекту	Швидкодія, безпека, простота

Отже, було визначено стратегію позиціонування, а саме визначено основні вимоги до товару цільової аудиторії: коректна робота, надійність, підтримка; базову стратегію розвитку: диференціація; ключові конкурентоспроможні позиції стартап- проекту: емоційність, адже є ключовою технологією і перевагою нашого проекту. Також сформовано комплексну позицію проекту: швидкодія, безпека, простота.

4.5 РОЗРОБКА МАРКЕТИНГОВОЇ ПРОГРАМИ

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у табл. 4.18 потрібно підсумувати результати попереднього аналізу конкурентоспроможності товару.

Таблиця 4.18 – Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Автоматизація певного бізнес-процесу: користувацької підтримки та відділу продажів	Не потрібно наймати додаткових фахівців для консультацій користувачів	Ширший спектр аналітики обраних рішень

Надалі розробляється трирівнева маркетингова модель товару (табл. 4.19).

Таблиця 4.19 – Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові

<table>
<tr><td>I. Товар за задумом</td><td colspan="3">Автоматизація певного бізнес-процесу: роботи складів та відділу продажів. Внаслідок цього не потрібно наймати додаткових фахівців для консультацій користувачів. Також це допомагає швидше та ефективніше працювати з клієнтами.</td></tr>
<tr><td rowspan="4">II. Товар у реальному виконанні</td><td>Властивості/ характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
<tr><td>Надійність
Емоційність
Найдовший час відповіді</td><td>М М М</td><td>Тл Тл Тл</td></tr>
<tr><td colspan="3">Оптимізує роботу магазинів, дає рекомендації щодо управління складами та стратегій ціноутворення</td></tr>
<tr><td colspan="3">Мобільний-додаток, на який заходять клієнти</td></tr>
<tr><td rowspan="2">III. Товар із підкріпленням</td><td colspan="3">Акції, знижки</td></tr>
<tr><td colspan="3">Підтримка</td></tr>
<tr><td colspan="4">За рахунок чого потенційний товар буде захищено від копіювання: патент.</td></tr>
</table>

Наступним кроком є визначення цінових меж (табл. 4.20). Аналіз проводиться експертним методом.

Таблиця 4.20 – Визначення меж встановлення ціни

№ п/п	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	125 грн. / 1000 викликів	250 грн. / 1000 викликів	>100000 грн. / місяць	275 грн. – 500 грн. / 1000 викликів

Визначено межі встановлення ціни на мобільний додаток, а саме рівень цін на товари-замінники – 125 грн. за 1000 викликів програмного інтерфейсу, рівень цін на товари-аналоги 250 грн., рівень доходів цільової групи споживачів - 100000 грн, верхня та нижня межі встановлення ціни на товар – 275-500 грн. Аналіз був проведений експертним методом.

Наступним кроком є визначення оптимальної системи збуту, в межах якого приймається рішення (табл. 4.21).

Таблиця 4.21 – Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Клієнти зазвичай оцінять можливості платформи через безкоштовну пробну версію, а потім почнуть платити за кожні 1000 викликів сервісу протягом довгого часу	Фінансування витрат на функціонування каналу збуту, фінансування збутових операцій.	Канал нульового рівня	Проводити збут власними силами

		Обслуговування проданих товарів.		

Отже, було сформовано систему збуту у вигляді щомісячної оплати за певну кількість викликів сервісу, та буде включати користувацьку підтримку. Збут буде проводитися власними силами.

Далі розробляється концепція маркетингових комунікацій (табл. 4.22).

Таблиця 4.22 – Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Це інтернет-компанії, які надають перевагу новітнім технологіям	E-mail, письмові звернення до компаній, конференції	Унікальна особливість чат- ботів - емоційність, покращує задоволення клієнтів	Привернути максимум уваги до продукту, змусити компанію спробувати продукт	Користувачі хочуть спілкуватись з живими людьми, компанії хочуть наймати менше людей - ми компроміс.

Отже, було розроблено концепцію маркетингових комунікацій: ключові позиції для позиціонування – емоційність розроблених чат-ботів

при спілкуванні, основне завдання рекламного повідомлення - привернути максимум уваги до продукту, змусити компанію спробувати продукт, з фокусуванням на концепцію рекламного звернення – компроміс між потребами компаній та їх користувачів.

4.6 ВИСНОВКИ ДО РОЗДІЛУ

У даному розділі були досліджені основні аспекти виходу на ринок мобільного додатку побудованого за допомогою комбінованої архітектури. Описаний продукт є доцільний для користувачів та компаній, які бажають оптимізувати роботу свого мобільного застосунку.

В рамках розділу було визначено перелік слабких, сильних та нейтральних характеристик та властивостей ідеї потенційного товару, що є підґрунтям для формування його конкурентоспроможності; обрана технологія реалізації ідеї проекту: для створення комбінованої архітектури системи. Ця архітектура є безкоштовною, зручною та гнучкою для експлуатації. Був проведений ступеневий аналіз конкуренції на ринку, SWOT аналіз та обґрунтовані фактори конкурентоспроможності. Також було обрано основні цільові групи: компанії, які потребують оптимізувати свої мобільні додатки, адже ця група мають схожі вимоги та готові сприйняти продукт. Його основні характеристики: новизна, надійність. Технологія буде захищена від копіювання за рахунок патенту.

Отже відповідно до проведених досліджень:

- існує можливість ринкової комерціалізації проекту;
- існують перспективи впровадження з огляду на потенційні групи клієнтів, бар’єри входження не є високими;

- проект має дві значні переваги перед конкурентами: кросплатформність, широкий спектр послуг аналітики обраних рішень;
- подальша імплементація є доцільною.

ВИСНОВКИ

Розробка масштабованих і стійких додатків є складною технічною задачею, в якій необхідно врахувати не лише базові вимоги до продукту, а й закласти можливості до розширення функціоналу, швидкої розробки нового функціоналу, масштабованості, повторного використання компонентів та до розширення команди інженерів, які були б спроможні працювати над застосунком.

Для вирішення цих задач було проаналізовано однонапрямлені та двонапрямлені архітектури. Основна проблема двонапрямлених архітектур – у втрачанні контролю над кінцевим станом системи, що призводить до можливості отримання невизначених станів програми. Це призводить до більш складної відладки програми, а також до створення неявних сайд-ефектів. Однонапрямовані архітектури дозволяють уникнути цих проблем. Але їх недоліком є проблеми продуктивності. Уся програма може надто часто змінювати стан і запускати занадто багато оновлень, які надсилаються підписникам. А це, в свою чергу, може знизити продуктивність застосунку. Наступна проблема класичної днонапрямованої архітектури – один глобальний стан. Щойно програма стане достатньо великою, з’ясувати будь-що з єдиного стану неминуче стане величезною проблемою.

На основі аналізу існуючих проблем та нових вимог було запропоновано створити комбіновану архітектуру. Слід відмітити, що запропонована модель є більш високорівневою, ніж її аналоги, має більш докладний опис шарів, в ній також пропонується поділ застосунку на кілька «ідейних» шарів. Серед запропонованих шарів наявні Feature-шари, за допомогою яких буде можливо відокремлювати функціонал застосунку та App-Independent шар, що надає можливість виносити загальний код, який не стосується конкретного функціоналу, у окремі модулі. Ці модулі також можуть бути спільними, наприклад, для компанії.

При розподілі застосунка на шари використовувались основні принципи проектування програмного забезпечення, серед яких: Low Coupling та High Cohesion, контроль інтерфейсу сутності, принципи SOLID, а також впровадження залежностей. За допомогою створення більш високорівневої моделі було отримано можливість використовувати будь-які архітектурні моделі будь-то MVC, MVVM або UDF архітектури.

Щодо UDF подібних архітектур було запропоновано декомпозувати єдиний глобальний Стан на під-стани, а також створити порівняння цих станів, аби уникнути випадків оновлення відображення, коли це не потрібно. Дані покращення за фактом є самостійними покращеннями, які можливо використовувати навіть без комбінованої архітектури.

Проблема пришвидшення розробки нового функціоналу за підтримки різних версій iOS була вирішена за допомогою створення DSL фреймворка над стандартними компонентами відображення для ОС iOS. При його розробці також були використанні розглянуті вище загальні архітектурні принципи проектування програмного забезпечення. Також, слід відмітити, що основні інтерфейси створювались подібними до більш нового нативного рішення SwiftUI. Це надає можливість доволі швидко замінити створений фреймворк на нативний, коли це дозволить мінімальна підтримуєма версія ОС. Створення даного фреймворку надало змогу описувати UI у декларативному стилі, що допомогає швидше та зручніше створювати нові модулі. На етапі розробки було проведено статичне тестування застосунку: обрано статичний аналізатор коду (лінтер). Після розробки додатку було проведено динамічне тестування застосунку, в ході якого було перевірено: правильне функціонування компонентів відображення за допомогою автоматизованого тестування. Усі вимоги були протестовані на смартфоні iPhone 11 Pro з операційною системою iOS 16.0. Результати

тестування показали, що створений фреймворк повністю відповідає сформульованим до нього вимогам.

В якості перспективи розвитку цієї теми передбачається створення більше практичних прикладів реалізації комбінованої архітектури для уникання невірного ідейного інтерпретування, а також подальший розвиток декларативного фреймворку та створення окремного фреймворку для декларативної навігації по застосунку.

ДЖЕРЕЛА

1. Походження Model View Controller [Електронний ресурс]. — Режим доступу до ресурсу: https://medium.com/@duncandevs/origins-of-model-view-controller-d685528857ce
2. Apple MVC [Електронний ресурс]. — Режим доступу до ресурсу: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1995
4. Архітектурний шаблон Model View Controller [Електронний ресурс]. — Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Model–view–presenter
5. Архітектура MVP [Електронний ресурс]. — Режим доступу до ресурсу: https://medium.datadriveninvestor.com/model-view-presenter-mvp-5c3439227f83
6. Архітектура MVP у iOS [Електронний ресурс]. — Режим доступу до ресурсу: https://saad-elaoulladi.medium.com/ios-swift-mvp-architecture-pattern-a2b0c2d310a3
7. Архітектура MVVM [Електронний ресурс]. — Режим доступу до ресурсу: https://jeremybytes.blogspot.com/2012/04/overview-of-mvvm-design-pattern.html
8. Архітектура UDF [Електронний ресурс]. — Режим доступу до ресурсу: https://habr.com/ru/company/inDrive/blog/571394/

9. Архітектура UDF [Електронний ресурс]. — Режим доступу до ресурсу: https://habr.com/ru/company/inDrive/blog/576660/
10. R. C. Martin, *Clean Architecture - A Craftsman's Guide to Software Structure and Design*. Englewood Cliffs, NJ: Prentice Hall, 2017
11. Hexagonal architecture [Електронний ресурс]. — Режим доступу до ресурсу: http://alistair.cockburn.us/Hexagonal+architecture
12. Model-view-intent [Електронний ресурс]. — Режим доступу до ресурсу: https://cycle.js.org/model-view-intent.html
13. Onion architecture [Електронний ресурс]. — Режим доступу до ресурсу: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
14. The elm architecture [Електронний ресурс]. — Режим доступу до ресурсу: https://guide.elm-lang.org/architecture/
15. A. Hunt, D. Thomas, The Pragmatic Programmer, Addison-Wesley, 1999
16. Clean Architecture. [Електронний ресурс]. — Режим доступу до ресурсу: https://8thlight.com/blog/assets/posts/2012-08-13-the-clean-architecture
17. J. Coplien and G. Bjørnvig, *Lean Architecture: for Agile Software Development*. Wiley, 2011
18. Декларативна та імперативна парадигми . [Електронний ресурс]. — Режим доступу до ресурсу: https://dev.to/ruizb/declarative-vs-imperative-4a7l
19. Injection [Електронний ресурс]. — Режим доступу до ресурсу: https://github.com/krzysztofzablocki/Sourcery
20. Sourcery [Електронний ресурс]. — Режим доступу до ресурсу: https://github.com/krzysztofzablocki/Sourcery

21. Вступ до Sourcery [Електронний ресурс]. — Режим доступу до ресурсу: https://www.hackingwithswift.com/articles/85/introduction-to-sourcery
22. Фундаментальна теорія тестування [Електронний ресурс] – Режим доступу до ресурсу: https://habr.com/ru/post/549054/

ДОДАТОК

Під час побудови фреймворку було використано систему контроля версій Git, а саме Github (Рис. 8.1).

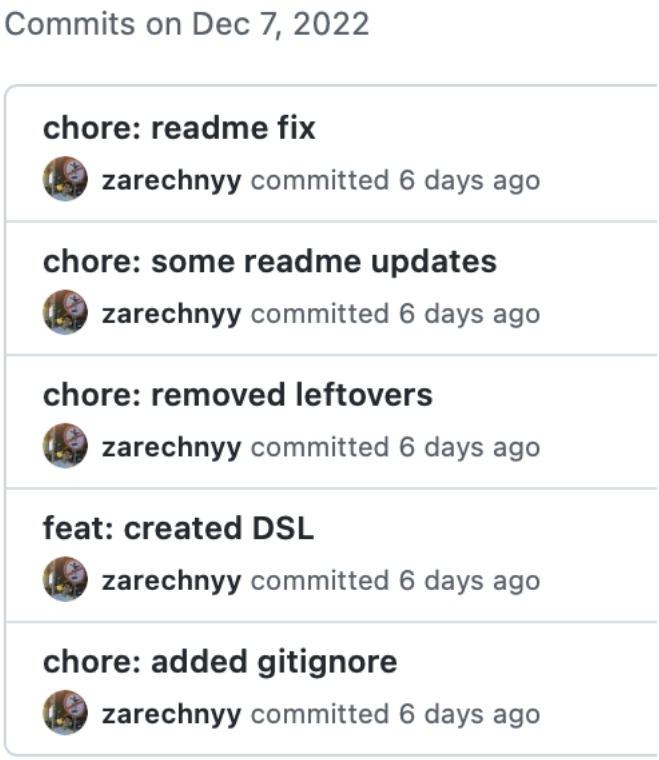
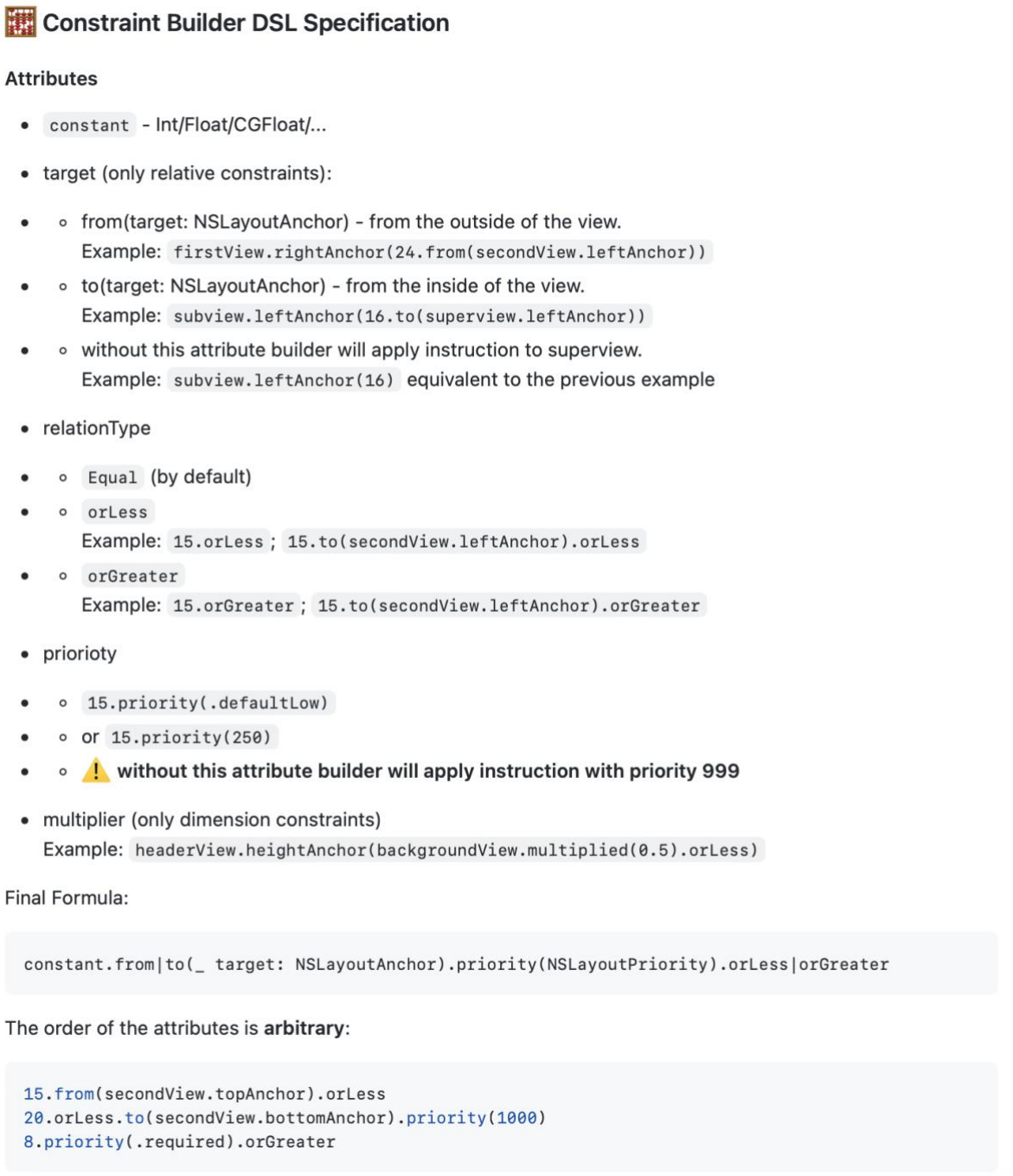


Рисунок 8.1 – Історія комітів

Також було описана документація для DSL фреймворка (Рис. 8.2).



Constraint Builder DSL Specification

Attributes

- `constant` - Int/Float/CGFloat/...
- target (only relative constraints):
  - from(target: NSLayoutAnchor) - from the outside of the view.
    Example: `firstView.rightAnchor(24.from(secondView.leftAnchor))`
  - to(target: NSLayoutAnchor) - from the inside of the view.
    Example: `subview.leftAnchor(16.to(superview.leftAnchor))`
  - without this attribute builder will apply instruction to superview.
    Example: `subview.leftAnchor(16)` equivalent to the previous example
- relationType
  - `Equal` (by default)
  - `orLess`
    Example: `15.orLess` ; `15.to(secondView.leftAnchor).orLess`
  - `orGreater`
    Example: `15.orGreater` ; `15.to(secondView.leftAnchor).orGreater`
- priorioty
  - `15.priority(.defaultLow)`
  - or `15.priority(250)`
  - ⚠️ **without this attribute builder will apply instruction with priority 999**
- multiplier (only dimension constraints)
  Example: `headerView.heightAnchor(backgroundView.multiplied(0.5).orLess)`

Final Formula:

```
constant.from|to(_ target: NSLayoutAnchor).priority(NSLayoutPriority).orLess|orGreater
```

The order of the attributes is **arbitrary**:

```
15.from(secondView.topAnchor).orLess
20.orLess.to(secondView.bottomAnchor).priority(1000)
8.priority(.required).orGreater
```

Рисунок 8.2 – Документація DSL фреймворка

Базовий інтерфейс для опису AutoLayout:

```
// AutoLauoutGuide.swift

public protocol AutoLauoutGuide {
    var leadingAnchor: NSLayoutXAxisAnchor { get }

    var trailingAnchor: NSLayoutXAxisAnchor { get }

    var leftAnchor: NSLayoutXAxisAnchor { get }

    var rightAnchor: NSLayoutXAxisAnchor { get }

    var topAnchor: NSLayoutYAxisAnchor { get }

    var bottomAnchor: NSLayoutYAxisAnchor { get }

    var widthAnchor: NSLayoutDimension { get }

    var heightAnchor: NSLayoutDimension { get }

    var centerXAnchor: NSLayoutXAxisAnchor { get }

    var centerYAnchor: NSLayoutYAxisAnchor { get }
}

extension UIView: AutoLauoutGuide {}

extension UILayoutGuide: AutoLauoutGuide {}
```

Моделі обгортки навколо системного API для опису Constraints:

```
// BidirectionalDimensionAutoLayoutAnchor.swift

public struct BidirectionalDimensionAutoLayoutAnchor {
```

```
    var relationType: RelationType

    var priority: UILayoutPriority

    struct Target {
        var layoutGuide: UILayoutGuide?
        var multiplier: CGFloat

        static var superview: Target {
            Target(layoutGuide: nil, multiplier: 1)
        }
    }

    var target: Target?

    var constant: CGFloat
}

public protocol BidirectionalDimensionAutoLayoutAnchorConvertible {
    func asBidirectionalDimensionAutoLayoutAnchor() -> BidirectionalDimensionAutoLayoutAnchor
}

extension BidirectionalDimensionAutoLayoutAnchor:
BidirectionalDimensionAutoLayoutAnchorConvertible {
    public func asBidirectionalDimensionAutoLayoutAnchor() ->
BidirectionalDimensionAutoLayoutAnchor { self }
}

public extension BidirectionalDimensionAutoLayoutAnchorConvertible {
    func multiplied(by multiplier: CGFloat) -> BidirectionalDimensionAutoLayoutAnchor {
        var copy = asBidirectionalDimensionAutoLayoutAnchor()

        assert(
            copy.target != nil,
            "Multiplier applied ONLY to target anchor attribute"
        )
```

```
        copy.target?.multiplier = multiplier

        return copy
    }

    func plus(_ constant: CGFloat) -> BidirectionalDimensionAutoLayoutAnchor {
        var copy = asBidirectionalDimensionAutoLayoutAnchor()

        copy.constant = constant

        return copy
    }

    func minus(_ constant: CGFloat) -> BidirectionalDimensionAutoLayoutAnchor {
        var copy = asBidirectionalDimensionAutoLayoutAnchor()

        copy.constant = -constant

        return copy
    }
}

public struct BidirectionRelativeAutoLayoutAnchor {
    var relationType: RelationType

    var priority: UILayoutPriority

    var target: AutoLauoutGuide?

    typealias Constant = RelativeAutoLayoutConstant

    var constant: Constant
}
```

```swift
public protocol BidirectionRelativeAutoLayoutAnchorConvertible {
    func asBidirectionalRelativeAutoLayoutAnchor() -> BidirectionRelativeAutoLayoutAnchor
}

extension BidirectionRelativeAutoLayoutAnchor: BidirectionRelativeAutoLayoutAnchorConvertible {
    public func asBidirectionalRelativeAutoLayoutAnchor() -> BidirectionRelativeAutoLayoutAnchor
{ self }
}

// DimensionAutoLayoutAnchor.swift

public struct DimensionAutoLayoutAnchor {
    var relationType: RelationType

    var priority: UILayoutPriority

    var target: Target?

    struct Target {
        var anchor: NSLayoutDimension?
        var multiplier: CGFloat

        static var superview: Target {
            Target(anchor: nil, multiplier: 1)
        }
    }

    var constant: CGFloat
}

public protocol DimensionAutoLayoutAnchorConvertible {
    func asDimensionAutoLayoutAnchor() -> DimensionAutoLayoutAnchor
}

extension DimensionAutoLayoutAnchor: DimensionAutoLayoutAnchorConvertible {
    public func asDimensionAutoLayoutAnchor() -> DimensionAutoLayoutAnchor { self }
```

```
}

public extension DimensionAutoLayoutAnchorConvertible {
    func multiplied(by multiplier: CGFloat) -> DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()

        assert(
            copy.target != nil,
            "Multiplier applied ONLY to target anchor attribute"
        )

        copy.target?.multiplier = multiplier

        return copy
    }

    func plus(_ constant: CGFloat) -> DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()

        copy.constant = constant

        return copy
    }

    func minus(_ constant: CGFloat) -> DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()

        copy.constant = -constant

        return copy
    }
}
```

Опис відношення Constraints у системі AutoLayout.

```
// RelationType.swift
```

```
enum RelationType {
    case less, equal, greater
}

public extension DimensionAutoLayoutAnchorConvertible {
    var orLess: DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()
        copy.relationType = .less
        return copy
    }

    var orGreater: DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()
        copy.relationType = .greater
        return copy
    }
}

public extension HorizontalRelativeAutoLayoutAnchorConvertible {
    var orLess: HorizontalRelativeAutoLayoutAnchor {
        var copy = asHorizontalRelativeAutoLayoutAnchor()
        copy.relationType = .less
        return copy
    }

    var orGreater: HorizontalRelativeAutoLayoutAnchor {
        var copy = asHorizontalRelativeAutoLayoutAnchor()
        copy.relationType = .greater
        return copy
    }
}

public extension VerticalRelativeAutoLayoutAnchorConvertible {
    var orLess: VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()
```

```
        copy.relationType = .less
        return copy
    }

    var orGreater: VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()
        copy.relationType = .greater
        return copy
    }
}

public extension BidirectionRelativeAutoLayoutAnchorConvertible {
    var orLess: BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()
        copy.relationType = .less
        return copy
    }

    var orGreater: BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()
        copy.relationType = .greater
        return copy
    }
}


public typealias HorizontalRelativeAutoLayoutAnchor =
RelativeAutoLayoutAnchor<NSLayoutXAxisAnchor>
public typealias VerticalRelativeAutoLayoutAnchor =
RelativeAutoLayoutAnchor<NSLayoutYAxisAnchor>

public struct RelativeAutoLayoutAnchor<Axis: AnyObject> {
    var relationType: RelationType

    var priority: UILayoutPriority
```

```
    var target: NSLayoutAnchor<Axis>?

    typealias Constant = RelativeAutoLayoutConstant

    var constant: Constant
}

public protocol HorizontalRelativeAutoLayoutAnchorConvertible {
    func asHorizontalRelativeAutoLayoutAnchor() -> HorizontalRelativeAutoLayoutAnchor
}

public protocol VerticalRelativeAutoLayoutAnchorConvertible {
    func asVerticalRelativeAutoLayoutAnchor() -> VerticalRelativeAutoLayoutAnchor
}

extension RelativeAutoLayoutAnchor: HorizontalRelativeAutoLayoutAnchorConvertible where Axis ==
NSLayoutXAxisAnchor {
    public func asHorizontalRelativeAutoLayoutAnchor() -> HorizontalRelativeAutoLayoutAnchor { self }
}

extension RelativeAutoLayoutAnchor: VerticalRelativeAutoLayoutAnchorConvertible where Axis ==
NSLayoutYAxisAnchor {
    public func asVerticalRelativeAutoLayoutAnchor() -> VerticalRelativeAutoLayoutAnchor { self }
}

// RelativeAutoLayoutConstant.swift

struct RelativeAutoLayoutConstant {
    var value: CGFloat

    enum Kind { case inset, offset }

    var kind: Kind

    static var zero: Self { Self(value: .zero, kind: .inset) }
}
```

```swift
// AutoLayoutConstant.swift

protocol AutoLayoutConstant: DimensionAutoLayoutAnchorConvertible,
                             HorizontalRelativeAutoLayoutAnchorConvertible,
                             VerticalRelativeAutoLayoutAnchorConvertible,
                             BidirectionRelativeAutoLayoutAnchorConvertible,
                             BidirectionalDimensionAutoLayoutAnchorConvertible {
    var value: CGFloat { get }
}

extension AutoLayoutConstant {
    public func asDimensionAutoLayoutAnchor() -> DimensionAutoLayoutAnchor {
        DimensionAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: nil,
            constant: value
        )
    }

    public func asHorizontalRelativeAutoLayoutAnchor() -> HorizontalRelativeAutoLayoutAnchor {
        HorizontalRelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: nil,
            constant: .init(value: value, kind: .inset)
        )
    }

    public func asVerticalRelativeAutoLayoutAnchor() -> VerticalRelativeAutoLayoutAnchor {
        VerticalRelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: nil,
            constant: .init(value: value, kind: .inset)
        )
```

```
    }

    public func asBidirectionalRelativeAutoLayoutAnchor() -> BidirectionRelativeAutoLayoutAnchor {
        BidirectionRelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: nil,
            constant: .init(value: value, kind: .inset)
        )
    }

    public func asBidirectionalDimensionAutoLayoutAnchor() ->
BidirectionalDimensionAutoLayoutAnchor {
        BidirectionalDimensionAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: nil,
            constant: value
        )
    }
}

extension Int: AutoLayoutConstant {
    public var value: CGFloat { CGFloat(self) }
}

extension Float: AutoLayoutConstant {
    public var value: CGFloat { CGFloat(self) }
}

extension Double: AutoLayoutConstant {
    public var value: CGFloat { CGFloat(self) }
}

extension CGFloat: AutoLayoutConstant {
    public var value: CGFloat { CGFloat(self) }
```

```
}

// HorizontalRelativeAutoLayoutAnchorConvertible.swift

public extension HorizontalRelativeAutoLayoutAnchorConvertible {
    func to(_ insetAnchor: NSLayoutXAxisAnchor) -> HorizontalRelativeAutoLayoutAnchor {
        var copy = asHorizontalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
            kind: .inset
        )

        copy.target = insetAnchor

        return copy
    }

    func from(_ offsetAnchor: NSLayoutXAxisAnchor) -> HorizontalRelativeAutoLayoutAnchor {
        var copy = asHorizontalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
            kind: .offset
        )

        copy.target = offsetAnchor

        return copy
    }
}

// VerticalRelativeAutoLayoutAnchorConvertible.swift

public extension VerticalRelativeAutoLayoutAnchorConvertible {
```

```
    func to(_ anchor: NSLayoutYAxisAnchor) -> VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
            kind: .inset
        )

        copy.target = anchor

        return copy
    }

    func from(_ anchor: NSLayoutYAxisAnchor) -> VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
            kind: .offset
        )

        copy.target = anchor

        return copy
    }
}

// BidirectionRelativeAutoLayoutAnchorConvertible.swift


public extension BidirectionRelativeAutoLayoutAnchorConvertible {
    func to(_ target: AutoLauoutGuide) -> BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
```

```
            kind: .inset
        )

        copy.target = target

        return copy
    }

    func from(_ target: AutoLauoutGuide) -> BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()

        copy.constant = RelativeAutoLayoutConstant(
            value: copy.constant.value,
            kind: .offset
        )

        copy.target = target

        return copy
    }
}

// NSLayoutDimension+Extensions.swift

extension NSLayoutDimension: DimensionAutoLayoutAnchorConvertible {
    public func asDimensionAutoLayoutAnchor() -> DimensionAutoLayoutAnchor {
        DimensionAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: DimensionAutoLayoutAnchor.Target(anchor: self, multiplier: 1),
            constant: .zero
        )
    }
}
```

```
extension NSLayoutXAxisAnchor: HorizontalRelativeAutoLayoutAnchorConvertible {
    public func asHorizontalRelativeAutoLayoutAnchor() -> HorizontalRelativeAutoLayoutAnchor {
        RelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: self,
            constant: .zero
        )
    }
}

extension NSLayoutYAxisAnchor: VerticalRelativeAutoLayoutAnchorConvertible {
    public func asVerticalRelativeAutoLayoutAnchor() -> VerticalRelativeAutoLayoutAnchor {
        RelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: self,
            constant: .zero
        )
    }
}

extension UIView: BidirectionRelativeAutoLayoutAnchorConvertible {
    public func asBidirectionalRelativeAutoLayoutAnchor() -> BidirectionRelativeAutoLayoutAnchor {
        BidirectionRelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: .almostRequired,
            target: self,
            constant: .zero
        )
    }
}


public extension DimensionAutoLayoutAnchorConvertible {
    func priority(_ value: UILayoutPriority) -> DimensionAutoLayoutAnchor {
```

```
        var copy = asDimensionAutoLayoutAnchor()

        copy.priority = value

        return copy
    }

    func priority(_ value: Float) -> DimensionAutoLayoutAnchor {
        var copy = asDimensionAutoLayoutAnchor()

        copy.priority = UILayoutPriority(value)

        return copy
    }
}

public extension VerticalRelativeAutoLayoutAnchorConvertible {
    func priority(_ value: UILayoutPriority) -> VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()

        copy.priority = value

        return copy
    }

    func priority(_ value: Float) -> VerticalRelativeAutoLayoutAnchor {
        var copy = asVerticalRelativeAutoLayoutAnchor()

        copy.priority = UILayoutPriority(value)

        return copy
    }
}

public extension HorizontalRelativeAutoLayoutAnchorConvertible {
    func priority(_ value: UILayoutPriority) -> HorizontalRelativeAutoLayoutAnchor {
```

```
        var copy = asHorizontalRelativeAutoLayoutAnchor()

        copy.priority = value

        return copy
    }

    func priority(_ value: Float) -> HorizontalRelativeAutoLayoutAnchor {
        var copy = asHorizontalRelativeAutoLayoutAnchor()

        copy.priority = UILayoutPriority(value)

        return copy
    }
}

public extension BidirectionRelativeAutoLayoutAnchorConvertible {
    func priority(_ value: UILayoutPriority) -> BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()

        copy.priority = value

        return copy
    }

    func priority(_ value: Float) -> BidirectionRelativeAutoLayoutAnchor {
        var copy = asBidirectionalRelativeAutoLayoutAnchor()

        copy.priority = UILayoutPriority(value)

        return copy
    }
}


public extension BidirectionalDimensionAutoLayoutAnchorConvertible {
```

```
    func priority(_ value: UILayoutPriority) -> BidirectionalDimensionAutoLayoutAnchor {
        var copy = asBidirectionalDimensionAutoLayoutAnchor()

        copy.priority = value

        return copy
    }

    func priority(_ value: Float) -> BidirectionalDimensionAutoLayoutAnchor {
        var copy = asBidirectionalDimensionAutoLayoutAnchor()

        copy.priority = UILayoutPriority(value)

        return copy
    }
}
```

Методи для створення Constaints у декларативному стилі:

```
// AutoLayoutItemConvertible.swift

public extension AutoLayoutItemConvertible {
    func layout(@ArrayBuilder<AutoLayoutItem> _ builder: @escaping (UIView) -> [AutoLayoutItem])
-> AutoLayoutItem {
        asAutoLayoutItem().layout({ view in
            for item in builder(view) { item.activate() }
        })
    }

    func topAnchor(_ anchor: VerticalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
        asAutoLayoutItem().layout({ view in
            makeRelativeConstraint(
                view: view,
                anchorProvider: { $0.topAnchor },
                second: anchor.asVerticalRelativeAutoLayoutAnchor(),
                invertConstant: false
            )
```

```
        })
    }

    func bottomAnchor(_ anchor: VerticalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
        asAutoLayoutItem().layout({ view in
            makeRelativeConstraint(
                view: view,
                anchorProvider: { $0.bottomAnchor },
                second: anchor.asVerticalRelativeAutoLayoutAnchor(),
                invertConstant: true
            )
        })
    }

    func centerYAnchor(_ anchor: VerticalRelativeAutoLayoutAnchorConvertible = 0) ->
AutoLayoutItem {
        asAutoLayoutItem().layout({ view in
            makeRelativeConstraint(
                view: view,
                anchorProvider: { $0.centerYAnchor },
                second: anchor.asVerticalRelativeAutoLayoutAnchor(),
                invertConstant: false
            )
        })
    }

    func verticalAnchor(_ anchor: BidirectionRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem
{
        topAnchor(anchor.toAxisAnchor(anchorPath: { $0.topAnchor }))
            .bottomAnchor(anchor.toAxisAnchor(anchorPath: { $0.bottomAnchor }))
    }

    func leftAnchor(_ anchor: HorizontalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
        asAutoLayoutItem().layout({ view in
            makeRelativeConstraint(
                view: view,
```

```
            anchorProvider: { $0.leftAnchor },
            second: anchor.asHorizontalRelativeAutoLayoutAnchor(),
            invertConstant: false
        )
    })
}

func leadingAnchor(_ anchor: HorizontalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem
{
    asAutoLayoutItem().layout({ view in
        makeRelativeConstraint(
            view: view,
            anchorProvider: { $0.leadingAnchor },
            second: anchor.asHorizontalRelativeAutoLayoutAnchor(),
            invertConstant: false
        )
    })
}

func rightAnchor(_ anchor: HorizontalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
    asAutoLayoutItem().layout({ view in
        makeRelativeConstraint(
            view: view,
            anchorProvider: { $0.rightAnchor },
            second: anchor.asHorizontalRelativeAutoLayoutAnchor(),
            invertConstant: true
        )
    })
}

func trailingAnchor(_ anchor: HorizontalRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
    asAutoLayoutItem().layout({ view in
        makeRelativeConstraint(
            view: view,
            anchorProvider: { $0.trailingAnchor },
            second: anchor.asHorizontalRelativeAutoLayoutAnchor(),
```

```
            invertConstant: true
        )
    })
}

func centerXAnchor(_ anchor: HorizontalRelativeAutoLayoutAnchorConvertible = 0) ->
AutoLayoutItem {
    asAutoLayoutItem().layout({ view in
        makeRelativeConstraint(
            view: view,
            anchorProvider: { $0.centerXAnchor },
            second: anchor.asHorizontalRelativeAutoLayoutAnchor(),
            invertConstant: false
        )
    })
}

func horizontalAnchor(_ anchor: BidirectionRelativeAutoLayoutAnchorConvertible) ->
AutoLayoutItem {
    leftAnchor(anchor.toAxisAnchor(anchorPath: { $0.leftAnchor }))
        .rightAnchor(anchor.toAxisAnchor(anchorPath: { $0.rightAnchor }))
}

func directionalHorizontalAnchor(_ anchor: BidirectionRelativeAutoLayoutAnchorConvertible) ->
AutoLayoutItem {
    leadingAnchor(anchor.toAxisAnchor(anchorPath: { $0.leadingAnchor }))
        .trailingAnchor(anchor.toAxisAnchor(anchorPath: { $0.trailingAnchor }))
}

func centerAnchor(_ anchor: BidirectionRelativeAutoLayoutAnchorConvertible) -> AutoLayoutItem {
    centerYAnchor(anchor.toAxisAnchor(anchorPath: { $0.centerYAnchor }))
        .centerXAnchor(anchor.toAxisAnchor(anchorPath: { $0.centerXAnchor }))
}

func edgesAnchors(
    _ insets: UIEdgeInsets = .zero,
```

```
    to target: UIView? = nil,
    priority: UILayoutPriority = UILayoutPriority(999)
) -> AutoLayoutItem {

    func make<Axis>(
        for anchor: (UIView) -> NSLayoutAnchor<Axis>,
        inset: CGFloat
    ) -> RelativeAutoLayoutAnchor<Axis> {
        RelativeAutoLayoutAnchor(
            relationType: .equal,
            priority: priority,
            target: target.map(anchor),
            constant: RelativeAutoLayoutAnchor<Axis>.Constant(value: inset, kind: .inset)
        )
    }

    return topAnchor(make(for: { $0.topAnchor }, inset: insets.top))
        .leftAnchor(make(for: { $0.leftAnchor }, inset: insets.left))
        .rightAnchor(make(for: { $0.rightAnchor }, inset: insets.right))
        .bottomAnchor(make(for: { $0.bottomAnchor }, inset: insets.bottom))
}

func widthAnchor(_ anchor: DimensionAutoLayoutAnchorConvertible) -> AutoLayoutItem {
    asAutoLayoutItem().layout({ (view: UIView) in
        makeDimensionConstraint(
            view: view,
            anchorProvider: \.widthAnchor,
            second: anchor.asDimensionAutoLayoutAnchor()
        )
    })
}

func heightAnchor(_ anchor: DimensionAutoLayoutAnchorConvertible) -> AutoLayoutItem {
    asAutoLayoutItem().layout({ (view: UIView) in
        makeDimensionConstraint(
            view: view,
```

```
            anchorProvider: \.heightAnchor,
            second: anchor.asDimensionAutoLayoutAnchor()
        )
    })
  }

  func sizeAnchor(_ anchor: BidirectionalDimensionAutoLayoutAnchorConvertible) ->
AutoLayoutItem {
    heightAnchor(anchor.toDimensionAnchor(anchorPath: { $0.heightAnchor }))
      .widthAnchor(anchor.toDimensionAnchor(anchorPath: { $0.widthAnchor }))
  }
}

private extension RelativeAutoLayoutAnchor.Constant {
  func finalized(invert: Bool) -> CGFloat {
    switch (kind, invert) {
    case (.inset, false): return value
    case (.inset, true): return -value
    case (.offset, false): return value
    case (.offset, true): return -value
    }
  }
}

private func makeDimensionConstraint(
  view: UIView,
  anchorProvider: (UIView) -> NSLayoutDimension,
  second anotherAnchor: DimensionAutoLayoutAnchor
) {
  let constraint: NSLayoutConstraint

  if let target = anotherAnchor.target {
    switch anotherAnchor.relationType {
    case .equal:
      constraint = anchorProvider(view).constraint(
        equalTo: target.anchor ?? anchorProvider(view.superview!),
```

```
                multiplier: target.multiplier,
                constant: anotherAnchor.constant
            )

        case .less:
            constraint = anchorProvider(view).constraint(
                lessThanOrEqualTo: target.anchor ?? anchorProvider(view.superview!),
                multiplier: target.multiplier,
                constant: anotherAnchor.constant
            )

        case .greater:
            constraint = anchorProvider(view).constraint(
                greaterThanOrEqualTo: target.anchor ?? anchorProvider(view.superview!),
                multiplier: target.multiplier,
                constant: anotherAnchor.constant
            )
        }
    } else {
        switch anotherAnchor.relationType {
        case .equal:
            constraint = anchorProvider(view).constraint(equalToConstant: anotherAnchor.constant)

        case .less:
            constraint = anchorProvider(view).constraint(lessThanOrEqualToConstant:
anotherAnchor.constant)

        case .greater:
            constraint = anchorProvider(view).constraint(greaterThanOrEqualToConstant:
anotherAnchor.constant)
        }
    }

    constraint.priority = anotherAnchor.priority
    constraint.isActive = true
}
```

```
private func makeRelativeConstraint<AnchorType>(
    view: UIView,
    anchorProvider: (UIView) -> NSLayoutAnchor<AnchorType>,
    second anotherAnchor: RelativeAutoLayoutAnchor<AnchorType>,
    invertConstant: Bool
) {
    let constraint: NSLayoutConstraint

    switch anotherAnchor.relationType {
    case .equal:
        constraint = anchorProvider(view).constraint(
            equalTo: anotherAnchor.target ?? anchorProvider(view.superview!),
            constant: anotherAnchor.constant.finalized(invert: invertConstant)
        )

    case .less:
        constraint = anchorProvider(view).constraint(
            lessThanOrEqualTo: anotherAnchor.target ?? anchorProvider(view.superview!),
            constant: anotherAnchor.constant.finalized(invert: invertConstant)
        )

    case .greater:
        constraint = anchorProvider(view).constraint(
            greaterThanOrEqualTo: anotherAnchor.target ?? anchorProvider(view.superview!),
            constant: anotherAnchor.constant.finalized(invert: invertConstant)
        )
    }

    constraint.priority = anotherAnchor.priority
    constraint.isActive = true
}

private extension BidirectionRelativeAutoLayoutAnchorConvertible {
    func toAxisAnchor<Axis>(
        anchorPath: (AutoLauoutGuide) -> NSLayoutAnchor<Axis>
```

```
    ) -> RelativeAutoLayoutAnchor<Axis> {
        let anhor = asBidirectionalRelativeAutoLayoutAnchor()

        return RelativeAutoLayoutAnchor(
            relationType: anhor.relationType,
            priority: anhor.priority,
            target: anhor.target.map(anchorPath),
            constant: anhor.constant
        )
    }
}

private extension BidirectionalDimensionAutoLayoutAnchorConvertible {
    func toDimensionAnchor(
        anchorPath: (AutoLauoutGuide) -> NSLayoutDimension
    ) -> DimensionAutoLayoutAnchor {
        let anchor = asBidirectionalDimensionAutoLayoutAnchor()

        return DimensionAutoLayoutAnchor(
            relationType: anchor.relationType,
            priority: anchor.priority,
            target: anchor.target.map({ DimensionAutoLayoutAnchor.Target(anchor:
$0.layoutGuide.map(anchorPath), multiplier: $0.multiplier) }),
            constant: anchor.constant
        )
    }
}

public protocol AutoLayoutItemConvertible: UIStackViewConfiguration {
    func asAutoLayoutItem() -> AutoLayoutItem
}

extension AutoLayoutItemConvertible {
    public func configure(stackView: UIStackView) {
        let item = self.asAutoLayoutItem()
        stackView.addArrangedSubview(item.view)
```

```
        item.move(to: stackView)
        item.activate()
    }
}
```

Абстракція для опису «Об'єкта відображення»

```
// AutoLayoutItem.swift

public final class AutoLayoutItem {
    init(view: UIView) { self.view = view }

    let view: UIView
    var constraintsContainer: [() -> Void] = []
    var afterBuild: () -> Void = {}

    func layout(_ build: @escaping (UIView) -> Void) -> AutoLayoutItem {
        constraintsContainer.append({ [view] in build(view) })
        return self
    }

    func move(to superview: UIView) {
        view.translatesAutoresizingMaskIntoConstraints = false
        superview.addSubview(view)
    }

    @discardableResult
    public func activate() -> UIView {
        for constraintBuilder in constraintsContainer { constraintBuilder() }
        constraintsContainer = []
        afterBuild()
        return view
    }
}
```

```swift
extension AutoLayoutItem: AutoLayoutItemConvertible {
    public func asAutoLayoutItem() -> AutoLayoutItem { self }
}


extension UIView: AutoLayoutItemConvertible {
    public func asAutoLayoutItem() -> AutoLayoutItem { AutoLayoutItem(view: self) }
}


public protocol AutoLayoutItemConvertible: UIStackViewConfiguration {
    func asAutoLayoutItem() -> AutoLayoutItem
}

extension AutoLayoutItemConvertible {
    public func configure(stackView: UIStackView) {
        let item = self.asAutoLayoutItem()
        stackView.addArrangedSubview(item.view)
        item.move(to: stackView)
        item.activate()
    }
}
```