НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
___________ С.Г. СТІРЕНКО
(підпис)
“__” _________2022 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS

Виконав : студент 4 курсу, групи ІП-84
(шифр групи)

Павловський Всеволод Євгенович __________
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, с.н.с., к.т.н. Антонюк А.І __________
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант нормоконтроль д.т.н., проф. Сімоненко В. П. __________
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

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

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент ___________
(підпис)

Київ – 2022 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
__________ С.Г. СТІРЕНКО
(підпис)
“__” _________2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Павловського Всеволода Євгеновича

1. Тема проєкту Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS

керівник проєкту Антонюк Андрій Іванович, доцент, с.н.с., к.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від «__» ______ 2022 року №_____

2. Термін здачі студентом закінченого проєкту __________ *2022р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області.

Розділ 2. Вибір і обґрунтування засобів реалізації системи.

Розділ 3. Деталі розробки системи.

Розділ 4. Використання та розришення розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «   »   2022 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2021 – 15.12.2021*	
2.	*Вивчення та аналіз завдання*	*15.12.2021 – 15.03.2022*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2022 – 25.03.2022*	
4.	*Розробка структур окремих підсистем*	*25.03.2022 – 5.04.2022*	
5.	*Програмна реалізація системи*	*5.04.2022 – 15.04.2022*	
6.	*Оформлення пояснювальної записки*	*15.04.2022 – 11.06.2022*	
7.	*Захист програмного продукту*	*29.05.2022*	
8.	*Передзахист*	*11.06.2022*	
9.	*Захист*	*24.06.2022*	

Студент-дипломник ________________ Всеволод ПАВЛОВСЬКИЙ
(підпис)

Керівник проєкту ________________ Андрій АНТОНЮК
(підпис)

АНОТАЦІЯ

У даній роботі було розглянуто підходи до створення віддалено налаштованих елементів інтерфейсу користувача. Було проаналізовано необхідність віддаленого налаштуваня інтерфейсу користувача, основні проблеми, та шляхи систематизації даного підходу до створення користувацького інтерфейсу клієнтських додатків. Розроблена система дає можливість зручно налаштовувати інтерфейс користувача додатку без необхідності публікувати нову версію програми, та зменшує можливість виникнення помилок, пов'язаних із людським фактором.

Ключові слова: інтерфейс користувача, мобільні додатки, операційна система iOS, Swift, SwiftUI.

ANNOTATION

This paper for a Bachelor's Degree describes approaches to creating remotely configured User Interface for mobile applications for iOS-based devices. The need for remote configuration of the user interface, the main problems, and ways to systematize this approach for client applications were analyzed. The developed system allows developers to easily configure UI of the application without the need to publish a new version of the program, and reduces the possibility of errors related to the human factor.

Keywords: User Interface, mobile applications, iOS operating system, Swift, SwiftUI.

Справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
	А4	***ІАЛЦ.467200.002 ТЗ***	Віддалене налаштування	4		
			користувацьких інтерфейсів			
			клієнтських додатків в			
			середовищі iOS			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Віддалене налаштування	64		
			користувацьких інтерфейсів			
			клієнтських додатків в			
			середовищі iOS			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Віддалене налаштування	1		
			користувацьких інтерфейсів			
			клієнтських додатків в			
			середовищі iOS			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Віддалене налаштування	1		
			користувацьких інтерфейсів			
			клієнтських додатків в			
			середовищі iOS			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Віддалене налаштування	1		
			користувацьких інтерфейсів			
			клієнтських додатків в			
			середовищі iOS			
			Алгоритм дій програмного			
			забезпечення			

Зм	***Лист***	***№ докум.***	***Підп***	***Дата***	***ІАЛЦ.467200.001 ОА***		
Розроб		Павловський В.Є.			*Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS* Опис альбому	Літ.	Аркуш	Аркушів
Перев							1	1
						КПІ ім. Ігоря Сікорського ФІОТ ІП-84		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS*»

Київ – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ 2

НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ 3

ПІДСТАВИ ДЛЯ РОЗРОБКИ 3

МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ 3

ДЖЕРЕЛА РОЗРОБКИ 3

ТЕХНІЧНІ ВИМОГИ 4

Вимоги до розробленого продукту 4

Вимоги до програмного забезпечення 4

Вимоги до апаратної частини 4

ЕТАПИ РОЗРОБКИ 4

ПЕРЕЛІК СКОРОЧЕНЬ

UI	(англ. User Interface) Інтерфейс користувача
UIKit	User Interface Kit
SwiftUI	Swift User Interface
JSON	JavaScript Object Notation
XML	Extensible Markup Language
ПЗ	Програмне забезпечення
ОС	Операційна система
RAM	(англ. Random-access memory) Оперативна пам’ять
DI	Dependency Injection
SPM	Swift Package Manager

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання поширюється на розробку системи для віддаленого налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування цієї системи є створення мобільних клієнтських додатків для операційної системи iOS.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом "Інформатики та обчислювальної техніки" кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка системи для віддаленого налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS із підходом, що пришвидшить процес доставки оновленого інтерфейсу користувача кінцевому споживачу, та зменшить ймовірність помилок, пов'язаних із людським фактором, при роботі із системою.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Надати систематичний підхід до створення віддалено налаштованих інтерфейсів користувача.
- Надати можливість користувачам розширювати функціонал системи.
- Надати користувачам системи простий і інтуїтивно-зрозумілий інтерфейс створення конфігурації елементів інтерфейсу користувача.
- Надати вичерпну та зрозумілу документацію для використання розробленої системи.

5.2. Вимоги до програмного забезпечення

- ОС MacOS 10.15 версії або вище.
- XCode 13 версії або вище.

5.3. Вимоги до апаратної частини

- MacBook, iMac 2016 року випуску або пізніше.
- RAM не менше ніж 8 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	*12.12.2021*
Вивчення та аналіз завдання	*01.03.2022*
Розробка архітектури та загальної структури системи	*01.03.2022*
Розробка структур окремих частин системи	*15.03.2022*
Програмна реалізація системи	*5.04.2022*
Виправлення помилок	*15.04.2022*
Оформлення пояснювальної записки	*11.06.2022*

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS»

Київ – 2022

ЗМІСТ

ВСТУП ... 4

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕНТНОЇ ОБЛАСТІ ... 6

1.1 Віддалена конфігурація ... 6

1.2 Операційна система iOS ... 8

1.3 Інтерфейс користувача мобільних додатків у середовищі iOS ... 9

1.4 Бібліотеки інтерфейсів користувача для створення мобільних додатків до операційної системи iOS ... 9

1.5 Віддалене налаштування користувацьких інтерфейсів ... 10

ВИСНОВОК ДО РОЗДІЛУ 1 ... 11

РОЗДІЛ 2. ВИБІР І ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ ... 12

2.1 Вимоги до системи ... 12

2.2 Мова програмування ... 14

2.2.1 Мова Objective-C ... 14

2.2.2 Мова Swift ... 14

2.2.3 Сторонні мови програмування ... 15

2.2.4 Вибір мови програмування для реалізації системи ... 15

2.3 Бібліотека для створення інтерфейсу користувача в середовищі операційної системи iOS ... 15

2.3.1 UIKit ... 16

2.3.2 SwiftUI ... 16

2.3.3 Вибір бібліотеки для створення інтерфейсу користувача ... 18

2.4 Форма структури даних налаштувань користувацького інтерфейсу у системі ... 18

2.5 Формат збереження і передачі конфігурації елементів користувацького інтерфейсу ... 20

2.5.1 JSON ... 20

2.5.2 XML ... 21

2.5.3 Безпосередньо бінарна дата 21

2.5.4 Вибір формату збереження і передачі конфігурації користувацького інтерфейсу 21

2.6 Сервіс віддаленої конфігурації 22

2.7 Редактор налаштувань користувацького інтерфейсу.................. 23

ВИСНОВОК ДО РОЗДІЛУ 2 25

РОЗДІЛ 3 ДЕТАЛІ РОЗРОБКИ СИСТЕМИ... 27

3.1 Розробка програмних компонентів бібліотеки елементів інтерфейсу користувача 27

3.1.1 Структура DecodableViewConfiguration 28

3.1.2 Протокол DecodableView 29

3.1.3 Протокол DecodableViewResolver 29

3.1.4 Клас DecodableViewsService... 30

3.1.5 Протокол DecodableViewModifier 31

3.1.6 Структура DefaultViewModifier 31

3.1.7 Стандартні елементи інтерфейсу користувача DefaultViewModifier... 32

3.1.8 Структура LabelView 32

3.1.9 Структура ImageView 34

3.1.10 Структура StackView 35

3.2 Розробка програмних компонентів редактору конфігурацій. 37

3.2.1 Структура JSONRow.. 38

3.2.2 enum JSONValue 39

3.2.3 Реалізація інтерфесу користувача... 41

3.2.3 Інтеграція розробленої бібліотеки DecodableUI у редактор конфігурацій 42

3.3 Налаштування сервісу віддаленої конфігурації.......................... 45

3.4 Інтеграція розробленої бібліотеки DecodableUI у клієнтські додатки.. 47

ВИСНОВОК ДО РОЗДІЛУ 3 49

РОЗДІЛ 4 ВИКОРИСТАННЯ ТА РОЗШИРЕННЯ РОЗРОБЛЕНОЇ СИСТЕМИ ... 50

4.1 Огляд редактору конфігурацій ... 50

4.2 Використання конфігурації елементів інтерфейсу користувача 54

4.3 Розширення системи ... 56

4.4 Рекомендації щодо розвитку та вдосконалення системи ... 59

ВИСНОВОК ДО РОЗДІЛУ 4 ... 60

ВИСНОВКИ ... 61

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 63

ДОДАТОК 1 ... 3

ДОДАТОК 2 ... 5

ДОДАТОК 3 ... 7

ВСТУП

Інтерфейс користувача — найважливіший компонент сучасних клієнтських додатків. Будь-який додаток має на меті виконання визначеного завдання на користь кінцевого користувача, тоді як інтерфейс відображає зв’язок користувача і функціоналу програми. Без даного зв’язку функціонал як незалежна одиниця не має сенсу. Якщо функціонал не має сенсу, можна вважати, що додаток не виконує своє призначення.

Підвищення досвіду користування додатком стає більш важливим з кожним роком. “Гіганти” технологічних компаній більше двох декад віддають пріоритет користувацькому досвіду, ніж функціоналу і технологіям [1]. Зростання конкуренції на ринку Інформаційних технологій показує, що кінцева аудиторія частіше надає перевагу тим сервісам, якими користуватись зручніше, аніж тим, які мають більшу швидкодію, або більш різноманітний функціонал.

Зі сторони технологій розробки мобільних додатків для пристроїв, працюючих на операційній системи iOS, конфігурація інтерфейсу має дві найбільші проблеми: оперативність доставки актуального інтерфейсу кінцевому користувачу та персоналізація користувацького інтерфейсу.

Оскільки публікація мобільного додатку для смартфонів на базі iOS здійснюється через централізований магазин додатків, кожне оновлення програми займає вагому кількість часу. Пов’язана ця затримка із переглядом додатку на відповідність правилам магазину, і може займати до 10 робочих днів [2].

Необхідність оперативності доставки актуального інтерфейсу кінцевому користувачу пояснюється потребою на оперативний доступ до актуальних даних і функціоналу; необхідність персоналізації користувацького інтерфейсу пов’язана із особливостями кінцевого користувача, оскільки кожна людина має

свої особливості і обмеження. Один із дієвих шляхів подолання обох проблем є віддалене налаштування інтерфейсу користувача. Для досконалої роботи даного підходу, необхідна певна система із набором правил, які зменшують можливість виникнення помилок у роботі програми, пов'язаних із людським фактором, а також забезпечення версатильності до конфігурованих даних. Моєю метою є створення прикладу даної системи, яка буде задовольняти основним факторам, наведеними вище.

Ця дипломна робота містить чотири розділи, що в кінцевому обсязі розкриють деталі створення системи для віддаленого налаштування інтерфейсів користувача клієнтських додатків в середовищі iOS та підходи її використання.

Перший розділ представляє собою огляд предметної області системи і основні шляхи її використання.

Другий розділ описує програмні засоби, які будуть використовуватись для вирішення задачі теми диплому, а також підходи до структуризації даних конфігурацій.

Третій розділ описує деталі реалізації розглянутої системи, набір правил до користувача для комплексної взаємодії компонентів системи між собою, проблеми, з якими довелось зіткнутися в ході розробки даної системи, і шляхи їх вирішення.

Четвертий розділ представляє собою огляд розробленої системи та її компонентів, приклади її використання для певних задач, розширення системи її користувачами для задоволення унікальних потреб, а також аналіз можливих вдосконалень розробленої системи та її компонентів.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Віддалена конфігурація

Віддалена конфігурація (англ. Remote Configuration) — це техніка розробки програмного забезпечення, при якій поведінка програми може бути віддалено змінена без необхідності публікації нової версії додатку. Зазвичай даний підхід реалізується створенням конфігурації “за замовчуванням”, яка буде використовуватись при первинному запуску програми та при відсутності доступу до віддаленого ресурсу конфігурації. Після встановлення налаштувань програми із первісної конфігурації, додаток завантажує актуальну конфігурацію із віддаленого ресурсу, і встановлює оперативні налаштування в режимі реального часу [3].

Концепт віддаленої конфігурації був популяризований компанією Google, завдяки її продукту Firebase, котрий являє собою хмарну платформу із різноманітними сервісами для мобільних додатків. З часом концепт був прийнятий на технологічному рівні більшістю компаній, які займаються розробкою мобільних додатків. На сьогоднішній день неможливо уявити великий за функціоналом мобільний додаток, який би не використовував у певній мірі техніку віддаленої конфігурації.

Важливим функціоналом техніки віддаленої конфігурації є персоналізація конфігурації під окрему групу користувачів. Такі групи можуть бути розділені між собою різноманітними факторами, як от: версія додатку, країна проживання користувача, вікова група або стать користувача, мова додатку та багатьма іншими характеристиками.

Окрім цього, сервіси віддалених конфігурацій в основному мають функціонал “A/B тестування” — тестування покращень додатку (в тому числі

серед певних груп користувачів), для збору аналітики валідації рівня задоволеності функціоналом. Надалі ця аналітика може мати важливий фактор при вирішенні доцільності певного функціоналу, або при виборі кращої версії серед декількох можливих.

З технологічної сторони, правилами завантаження віддаленої конфігурації лежать на стороні мобільного додатку. Таким чином, розробник клієнтського додатку обирає частоту звернення до сервісу віддаленої конфігурації, як використовувати отримані дані, в якому вигляді зберігати (при необхідності), та звідки брати початкову конфігурацію. В деяких випадках розробники мобільних додатків зберігають останню версіях віддаленої конфігурації у пам’яті клієнта і завантажують початкові данні саме із ресурсів пристрою користувача додатку.

Таким чином, віддалена конфігурація може містити дані налаштувань, які стосуються різних компонентів клієнтського додатку.

Найчастіше, віддалена конфігурація представлена у конкретному форматі обміну даних, які підпорядковуються певним специфікаціям кодування (на приклад: JSON, XML). Сама структура віддалених конфігурацій представлена у вигляді структури даних словник (“ключ-значення” або хеш-таблиця), де кожному ключу відповідає певний об’єкт даних. Таким чином, отримана клієнтом віддалена конфігурація розглядається програмою не одночасно. Певна ділянка коду, залежна від значення віддаленої конфігурації дістає об’єкт налаштувань безпосередньо по ключу, зменшуючи навантаження на систему. Об’єктом налаштування, в свою чергу, може бути як простий тип даних (значення правди, рядок, число), так і складний тип даних (масив або словник).

1.2 Операційна система iOS

iOS — операційна система для мобільних девайсів, розроблена компанією Apple ексклюзивно для пристроїв компанії. Операційна система iOS була створена під керівництвом Стіва Джобса і представлена у 2007 році на презентації, яку вважають найважливішою у сфері інформаційних технологій в історії[4]. Аналізуючи зміни на ринку мобільних пристроїв після представлення операційної системи iOS та першого iPhone, можна оцінити вплив даної розробки на технологічну сферу.

Окрім багатьох технологічних інновацій, компанія Apple задала новий шлях розвитку комунікації користувача із мобільними пристроями. Графічний інтерфейс, розрахований на дотики і жести, надав можливості забезпечити користувачу найбільш зручні способи комунікації із мобільними програмами. Сама операційна система iOS була розроблена на основі подібної до Unix екосистеми Darwin, яка відрізняється своєю захищеністю та надійністю. Оскільки основа Darwin є Unix, основні модулі операційної системи iOS написані на C та C\++, але, одночасно, і з використанням мови програмування Objective-C. Із розвитком системи, розробники компанії Apple зрозуміли недоліки мови програмування Objective-C, і убезпечили її за допомогою створення мови програмування Swift. Таким чином, статично-типізований та компільований Swift став основним шляхом розробки програм для операційних систем компанії Apple.

1.3 Інтерфейс користувача мобільних додатків у середовищі iOS

Графічний користувацький інтерфейс мобільного додатку у середовищі iOS — це один із шляхів взаємодії в реальному часі користувача із програмою, запущеною на його пристрої на базі операційної системи iOS. Компанія Apple має значну кількість правил і описів дизайну мобільного додатку, які, на їх думку, мають забезпечити безперешкодну взаємодію користувача із додатком розробника як частиною “екосистеми Apple”. Таким чином компанія полегшує розробникам задачу зі створення графічного інтерфейсу, який буде для користувача найбільш зручним і зрозумілим.

Окрім саме елементів інтерфейсу користувача, важливу роль у сприйнятті користувачем додатку є їх позиціонування. Елементи користувацького інтерфейсу мають бути розташовані у зручній для користувача частині екрану, бути видимими у відповідний момент часу, мати обраний розмір і вигляд: колір, стиль, шрифт тощо. Тому, інформація про елемент інтерфейсу користувача має бути представлена у таких структурах даних, які зберігають унікальні для окремого виду елементу інтерфейсу параметри, а також загальні параметри, які мають усі елементи інтерфейсу користувача (на приклад: ширина, колір фону тощо).

1.4 Бібліотеки інтерфейсів користувача для створення мобільних додатків до операційної системи iOS

Для розробки клієнтських додатків на базі iOS, MacOS та інших операційних систем, розроблених компанією, Apple надає фреймворки для взаємодії із життєвим циклом системи і створення інтерфейсів користувача.

Дані бібліотеки надають розробнику способи взаємодії із подіями життєвого циклу додатку, можливість створення екземплярів системних компонентів і їх модифікації, можливість надання їм певної логіки і задання поведінки взаємодії з ними користувача додатку і системи.

1.4 Віддалене налаштування користувацьких інтерфейсів

Під віддаленим налаштуванням користувацького інтерфейсу клієнтського додатку можна розуміти наступний концепт: віддалена конфігурація мобільного додатку описує інтерфейс користувача (або деякі його елементи) клієнтського додатку.

Даний підхід набуває популярності у компаніях-розробниках мобільних додатків, розрахованих на значну аудиторію та велику кількість груп (на приклад: компанія Meta [5]).

Проте, проаналізувавши більшість інформації стосовної віддаленого налаштування клієнтських додатків, можна помітити, що об’єкти налаштування, напрямлені на інтерфейс користувача, в основному стосуються окремих параметрів цих елементів інтерфейсу, але не існують як незалежний об’єкт, який описує елемент інтерфейсу в цілому.

В межах даної дипломної роботи варто виділити таке поняття як елемент віддаленої конфігурації користувацького інтерфейсу — структура даних, яка самостійно описує заданий елемент інтерфейсу користувача.

Таким чином, можна сформулювати поняття системи віддаленого налаштування інтерфейсу користувача — система підходів, засобів програмування, кодової бази і правил, які дозволяють налаштовувати інтерфейс користувача віддалено, за допомогою набору елементів віддаленої конфігурації користувацького інтерфейсу.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було проаналізовано предметну область системи, ключові терміни та область застосунку системи. Були розглянуті і пояснені основні моменти, необхідні для розуміння області застосування системи, а також, проблеми і необхідність рішення, яке може їх розв'язати. Було розглянуто поняття інтерфейсу користувача, основні його елементи і властивості, та визначено його важливість у сучасній розробці клієнтський додатків. Було розглянуто поняття віддаленої конфігурації, описано основні фактори даної, методи і функціонал які зобов'язані задовільняти цьому підходу. Було описано ознаки і якості середовища розробки проєкту – операційної системи iOS, а також, особливості розробки мобільних додатків для цієї операційної системи. Було розглянуто поняття бібліотеки графічного інтерфейсу, які слугують для створення інтерфейсів користувача мобільних додатків на базі операційної системи iOS. Також, було сформовано поняття віддалено налаштованого інтерфейсу користувача клієнтських додатків, і описано в яких ситуаціях даний підхід є доцільним.

В області розробки даного проєкту – створення комплексної системи для віддаленого налаштування інтерфейсу користувача клієнтських додатків в середовищі iOS і всіх програмних компонентів, які реалізують поставлену проєктом задачу і виконують усі для цього вимоги. Комплексний підхід у вирішені задачі цього дипломного проекту зменшує ймовірність виникнення помилок, пов'язаних із людським фактором, і збільшує надійсність системи.

РОЗДІЛ 2. ВИБІР І ОБҐРУНТУВАННЯ ЗАСОБІВ РЕАЛІЗАЦІЇ СИСТЕМИ

2.1 Вимоги до системи

Виходячи із висновків аналізу предметної області, можна окреслити наступні вимоги до системи віддаленого налаштування інтерфейсу користувача для мобільних додатків в середовищі iOS:

1. Система має забезпечувати достатній рівень надійності і мінімізувати можливість помилок, що виникають внаслідок людського фактору.
2. Система має забезпечити можливість персоналізації інтерфейсу користувача в залежності від обраних критеріїв.
3. Система має забезпечити оперативну конфігурацію інтерфейсу для мобільного додатку у кожний період часу.
4. Елемент конфігурації має містити достатньо інформації для створення екземпляру одиниці графічного інтерфейсу та його налаштування.

Перший пункт вимог до системи має на увазі забезпечення розробникам і людям, відповідальним за створення налаштувань віддаленої конфігурації, зручного і наглядного шляху для створення налаштувань та їх валідації перед завантаженням до сервісу віддаленої конфігурації. Дану вимогу можна виконати, обравши редактор налаштувань, який задовольняє наступним вимогам:

1. Редактор мусить надавати зручний шлях для створення конфігурації у заданому форматі.
2. Редактор мусить мати можливість перегляду результату конфігурації у реальному часі.

3. Редактор мусить мати можливість збереження створених налаштувань для їх завантаження до сервісу віддаленої конфігурації.

Таким чином, даний пункт вимагає від системи збереження кодової бази, яка відповідає за декодування інтерфейсу користувача із конфігурації у безпосередньо одиниці графічного інтерфейсу та його налаштування, у окрему бібліотеку (надалі, бібліотека декодування графічного інтерфейсу).

Другий пункт вимог до системи являє собою здебільшого вимогу до сервісу віддаленої конфігурації, і правила комунікації мобільного додатку з даним сервісом.

Третій пункт вимог до системи відноситься до шляхів отримання мобільним додатком віддалених налаштувань із сервісу віддаленої конфігурації.

Четвертий пункт вимог до системи має на увазі аналіз і розроблення структури налаштувань, які зумовлять виконання даного пункту.

Проаналізувавши вимоги до системи, можна виділити наступні групи засобів реалізації:

1. Мова програмування для реалізації системи. Мається на увазі мова програмування бібліотеки декодування графічного інтерфейсу та мова програмування додатків, які використовують дану систему.
2. Бібліотека для створення інтерфейсу користувача в середовищі операційної системи iOS.
3. Форма структури даних налаштувань користувацького інтерфейсу.
4. Формат збереження і передачі конфігурації користувацького інтерфейсу.
5. Сервіс віддаленої конфігурації.
6. Редактор налаштувань користувацького інтерфейсу.

2.2 Мова програмування

Нативне написання програм для операційної системи iOS обмежене лише двома мовами програмування: Objective-C і Swift. Інші мови програмування використовуються виключно при розробці додатків в якості допоміжних виконувальних кодових одиниць на етапі збірки додатку, або, інтерпретуються сторонніми продуктами у нативний додаток. Таким чином, при розгляді не нативних мов програмування, важливо розуміти їх у сукупності із системою інтерпретатора.

Найбільш популярними сторонніми мовами програмування додатків для операційної системи iOS є JavaScript, Flutter, Dart та ін.

Далі розглянуто і проаналізовано можливі варіанти вибору мови програмування для реалізації програмних компонентів системи.

2.2.1 Мова Objective-C

Мова програмування Objective-C базується на мові C, із додаванням принципів об'єктно орієнтованого програмування і динамічного життєвого циклу. Розроблена компанією Next та принесена у компанію Apple Стівом Джобсом, мова програмування Objective-C довгий час була основним шляхом для розробки додатків для операційних систем iOS та MacOS. Objective-C має проблеми з безпечністю через свою динамічну природу.

2.2.2 Мова Swift

Swift — це мультипарадигмова, компільована та статично-типізована мова програмування, розроблена компанією Apple як краща альтернатива до мови

програмування Objective-C. Мова програмування Swift має значні переваги у швидкодії над Objective-C (2.6 разів швидше) та Python (8.4 разів швидше) [6].

Більшість сучасних фреймворків для розробки додатків до операційних систем iOS, MacOS та iPadOS базуються на мові програмування Swift.

2.2.3 Сторонні мови програмування

JavaScript, Flutter, Dart та Kotlin широко використовуються для написання крос-платформних додатків для операційних систем iOS та Android.

Основний підхід до написання додатків, використовуючи перелічені мови програмування, є інтерпретація коду до Swift або Objective-C з подальшою інтерпретацією у машинний код. Таким чином, швидкодія компіляції додатків, написаних на цих мовах програмування, нижча, ніж у додатків, написаних за допомогою нативних інструментів [7]. Окрім цього, інтерпретований доступ до системи значно зменшує кастомізацію інструментів, у порівнянні із нативними.

2.2.4 Вибір мови програмування для реалізації системи

Зважаючи на недоліки та переваги розглянутих вище варіантів, остаточний вибір зроблено в сторони мови програмування Swift через її швидкодію, широкий доступ до системи і її комонентів, та надійність.

2.3 Бібліотека для створення інтерфейсу користувача в середовищі операційної системи iOS

Компанія Apple надає розробникам вибір між двома фреймворками для створення інтерфейсу користувача додатків для операційної системи iOS —

UIKit та SwiftUI. Ці кодові бази не тільки дають доступ до системних елементів, а їх використання також задає принципи і правила життєвого циклу додатку.

Далі розглянуто бібліотеки для створення інтерфейсу користувача в середовищі операційної системи iOS UIKit та SwiftUI, та вибір найбільш доцільної для використання у проекті.

2.3.1 UIKit

Фреймворк UIKit (User Interface Kit) використовується для створення інтерфейсів користувача у середі iOS. UIKit надає розробнику об'єкт інтерфесу *UIView (Class)* та його сабкласи, які використовуються для створення, композиції та відмальовування елементів UI, та делегує методи спілкування із користувачем додатку.

Використання компонентів інтерфейсу у UIKit має основу імперативного програмування. При використанні фреймворку UIKit використовується архітектурний патерн MVC.

В наборі UI елементів фреймворку UIKit присутні основні елементи інтерфейсу користувача, такі як поля для вводу текстових даних, зображення, таблиці тощо. Із елементів стилізації цих компонентів, UIKit надає інтерфейси шрифтів, кольору, форми і т.п.

Фреймворк UIKit є основним і найпопулярнішим рішенням для створення мобільних додатків для операційної системи iOS.

2.3.2 SwiftUI

SwiftUI — крос-платформне рішення для створення користувацького інтерфейсу для операційних систем iOS, MacOS, iPadOS та WatchOS.

Код написаний із використанням SwiftUi інтерпретується в елементи відповідного до системи запуску додатку (в середовищі iOS це UIKit).

Фреймворк SwiftUI надає користувачу доступ до генерік типу *View (Struct)* і відповідних елементів стилізації елементів (*Color*, *Font*, *Shape* тощо).

Підтримка останньої версії фреймворку SwiftUI дуже обмежена: для версій операційної системи iOS – це 14 версія і вище; для MacOS – це 10.15 версія і вище. Такі обмеження підтримки зумовлють низький відсоток використання у комерційних продуктах. Проте популярніть фреймворку зростає у геометричній прогресії із виходом кожної нової версії операційних систем Apple (див. рис. 2.1) [8].

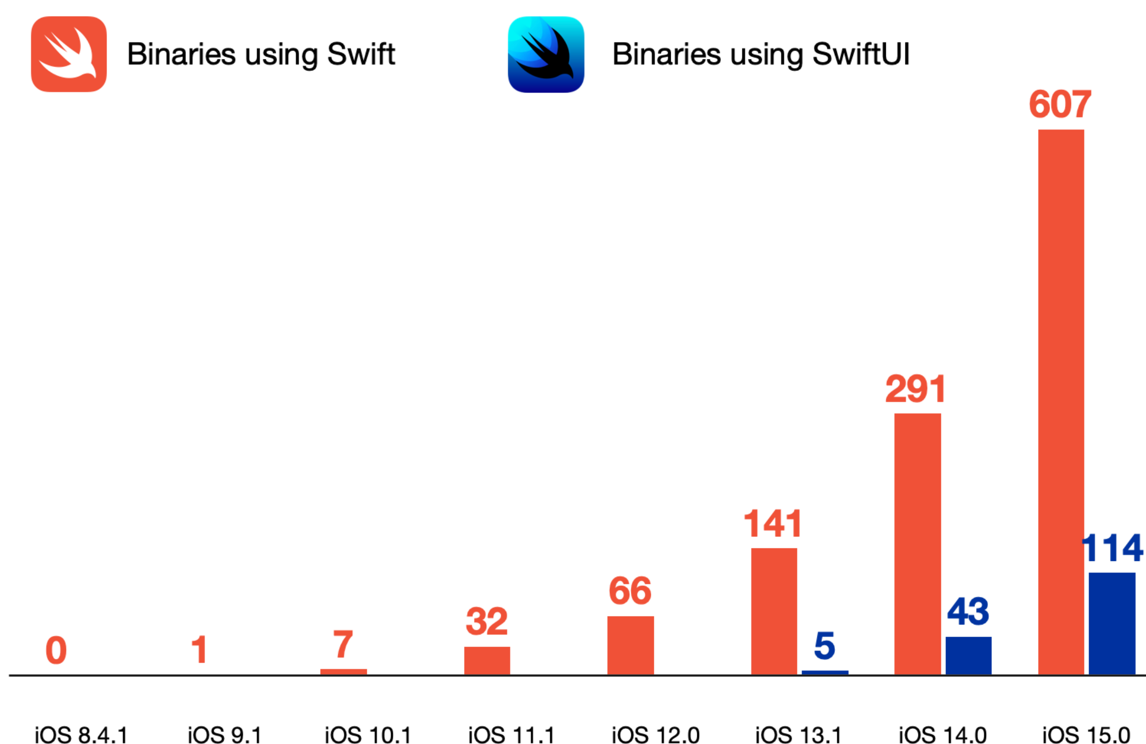


Рисунок 2.1 – кількість бінарних файлів з використанням фреймворку SwiftUI в компонентах операційної системи iOS в залежності від версії.

SwiftUI має за основу декларативний підхід до написання інтерфейсу користувача. Тому створення і налаштування елементів інтерфейсу користувача легше реалізується за допомогою фреймворку SwiftUI. Також декларативна основа фреймворку зумовлює більшу швидкодію на етапі компіляції, а використання типу даних *Struct* в основі елементів покращує

надійність системи, мінімізує вірогідність витоку пам’яті у життєвому циклі додатку, та зумовлює швидше читання даних з пам’яті.

2.3.2 Вибір бібліотеки для створення інтерфейсу користувача

Проаналізувавши варіанти бібліотек для створення елементів інтерфейсу користувача та їх переваги і недоліки, вибір було зроблено в сторону фреймворку SwiftUI.

Використання даного фреймворку пришвидшує розробку і швидкодію системи, і дає змогу розробляти більш зрозумілі для програміста елементи інтерфейсу і шляхи до взаємодії з ними. Окрім цього, елементи інтерфейсу користувача у даному фреймворку, мають подібну систему стилізації, що дає змогу уніфікувати загальні стилі елементів, і спростити структуру конфігурації UI елементів у системі.

2.4 Форма структури даних налаштувань користувацького інтерфейсу у системі

Оскільки елемент конфігурації має містити достатньо інформації для створення екземпляру одиниці графічного інтерфейсу та його налаштування, можна виділити наступні критерії до форми структури даних налаштувань користувацького інтерфейсу:

1. Конфігурація повинна містити інформацію про елемент інтерфейсу користувача.
2. Конфігурація має забезпечувати дані для налаштування відповідного елементу інтерфейсу користувача в системі.

Перший критерій можна задовольнити збереженням у конфігурації назви структури чи класу для подальшого створення екземпляру елементу інтерфейсу користувача.

Другий критерій можна задовольнити збереженням у конфігурації параметрів, необхідних для налаштування відповідного елементу інтерфейсу користувача у системі. Кожний елемент інтерфейсу користувача може мати унікальні параметри, такі як: посилання на зображення для елементу інтерфейсу типу зображення на екрані, текст для елементу типу екранний текст, унікальні дані власних елементів, створених користувачами системи тощо; та спільні параметри, такі як: ширина у точках (*CGPoint* для обраної бібліотеки SwiftUI) на екрані, висота у точках, колір фону, заокруглення елементу тощо.

Виходячи із критеріїв, описаних вище, можна зробити висновок щодо типу структури даних для збереження налаштувань у системі. У мові програмування Swift такі дані можна представити лише за допомогою типів *Dictionary* (словник), *Class*(клас), або *Struct* (структура).

Тип Dictionary не зовсім задовольняє умовам статичної типізації у мові програмування Swift, оскільки не може встановлювати чіткі обмеження до даних, і вимагає більше уваги розробника для уникнення помилок у життєвому циклі додатку і коду для реалізації перетворення записів у елементи інтерфейсу користувача бібліотеки SwiftUI.

Типи *Class* і *Struct* є більш доцільними у даному випадку. Основною різницею між цими двома типами є шлях доступу до значень, збережених у пам’яті під їх змінними: тип *Class* є типом посилання (*Reference type*), а *Struct* — типом значення (*Value type*). Оскільки дані після отримання не підлягають особливим змінам, а також тип Class може причинити витік пам’яті у життєвому циклу додатку, вибір було зроблено у сторону типу Struct.

Проаналізувавши критерії і можливі типи даних для задоволення поставленої задачі, форму структури даних налаштувань користувацького інтерфейсу у системі можна описати наступним чином:

1. Тип зберігаємих даних: *Struct*.
2. Клас елементу інтерфейсу: *String* (рядок)
3. Параметри для налаштування елементу інтерфейсу: поля значень у структурі даних типу Struct, із відповідними полями унікальних та загальних параметрів налаштування і стилізації.

2.5 Формат збереження і передаці конфігурації елементів користувацького інтерфейсу

Далі розглянуто і проаналізовано можливі варіанти формату збереження і передачі конфігурації елементів користувацького інтерфейсу.

2.5.1 JSON

JSON — формат обміну даних, який представляє специфікацію кодування даних. Формат JSON виділяється від інших своєю оптимізацією розміру збережених даних і є одним із найбільш простих форматів для кодування і розкодування. Це пояснюється поширеністю підтримки формату JSON більшістю сучасних мов програмування.

JSON формат базується на текстовому представлені і представляє структуровані дані. Синтаксис опису структур даних у JSON форматі ґрунтується на синтаксисі мови програмування JavaScript.

Формат JSON представляє собою єдиний об’єкт даних (словник), де кожному ключу відповідає один із основних видів даних (рядок, число, масив, чи об’єкт).

Роботу із форматом JSON можна представити у наступній послідовності:

1. Дані кодуються у JSON-форматний рядок тексту.

2. Дані у вигляді рядку передаються по відповідному протоколу отримувачу.
3. Дані формату JSON у вигляді рядку декодуються реципієнтом у відповідні структури даних.

2.5.2 XML

XML — формат обміну даних, який являє собою мову розмітки, описуючу форму даних і визначаючу набір правил для кодування документів у формат, який зрозумілий одночасно людиною і машиною.

XML формат базується на створенні “тегів“, необхідних для опису даних згідно певних структур, необхідних для виконання вимог користувача.

XML є документно-орієнтованим форматом обміну даних, на відміну від формату JSON та інших об’єктно-орієнтованих форматів.

2.5.3 Безпосередньо бінарна дата

Мова програмування Swift дає можливість перетворювати дані представлені у вигляді простих чи складних типів даних у бінарну дату, і навпаки.

Такі дані мають перевагу у швидкодії при кодуванні і розкодуванні, проте, даний вигляд збереження даних не є зрозумілим для користувача.

2.5.4 Вибір формату збереження і передачі конфігурації користувацького інтерфейсу

Формат JSON і бінарна дата мають перевагу перед XML у зберіганні даних відповідно до обраної структури даних налаштувань з огляду на об’єктну

орієнтованість цих форматів та простоту при їх розкодуванні в обраній мові програмування Swift.

Не зважаючи на більшу швидкодію при розкодуванні бінарної дати, остаточним вибором є формат JSON, оскільки є зрозумілим для людини. Даний фактор зменшує вірогідність помилок системи, викликаних людським фактором.

Таким чином, конфігурація користувацького інтерфейсу в обраному форматі має наступний загальний вигляд:

```
{
  "type": "<Назва типу елементу конфігурації>",
  "parameters": {
    "<Назва змінної параметру>": <Значення параметру>
  }
}
```

Рисунок 3.2 – Загальний вигляд конфігурації елементу інтерфейсу в форматі JSON

2.6 Сервіс віддаленої конфігурації

Серед існуючих комерційних рішень для збереження віддаленої конфігурації є лише Firebase Remote Config.

Firebase Remote Config — хмарний сервіс, який дозволяє розробникам зберігати та публікувати віддалені налаштування. Серед додаткового функціоналу, Firebase Remote Config має гнучкі налаштування для персоналізації налаштувань відповідно до вибраних критеріїв та підтримку A/B тестів. Серед наявних планів у сервісах Firebase є безкоштовні варіанти, обмежені кількістю підключених клієнтів і розмірами команії-розробника[9]. Окрім цього, сервісами Firebase користуються більше 2.5 тис. компаній (серед

них Atlassian, Twitch, Glovo та багато інших) у розробках власних проектів[10]. Така спільнота користувачів зумовлює постійне покращення продукту від Google, а також, вказує на ступінь перевірки і надійності сервісу у різних сферах розробки програмних додатків.

Розробка власного сервісу віддаленої конфігурації коштує значну кількість людино-годин, необхідних на розробку і тестування, та грошових витрат, необхідних на розміщення сервісу у хмарі.

Зручність користування сервісом Firebase Remote Config, а також безкоштовний варіант використання сервісу без додаткового функціоналу у вигляді аналітики та пуш-сервісів, роблять Firebase Remote Config кращою альтернативою у порівнянні із розробкою власного сервісу віддаленої конфігурації.

2.7 Редактор налаштувань користувацького інтерфейсу

Оскільки форматом зберігання налаштувань інтерфейсу користувача було обрано формат JSON, можна виділити три підходи до редагування конфігурації:

1. Сторонній редактор коду.
2. Власний редактор коду.
3. Власний об’єктно-орієнтований редактор JSON формату.

Більшість сучасних редакторів коду мають підтримку JSON формату. Окрім цього, сучасні редактори коду оснащенні підсвіткою синтаксису і форматуванням. Проте, при створенні інтерфейсу користувача, використовуючи сторонні редактори коду, неможливо у реальному часі переглядати інтерпретований системою користувацький інтерфейс. Цієї проблеми можна уникнути, написавши власний редактор коду.

Об’єктно-орієнтований редактор JSON формату представляє собою підхід написання об’єктів формату JSON у зручному вигляді, зменшуючи можливість

виникнення помилок, пов’язаних із людським фактором. Крім того, за умови розробки власного об’єктно-орієнтовного редактору JSON формату, можна вбудувати у нього перегляд сконфігурованого інтерфейсу користувача у режимі реального часу.

Зважаючи на переваги, які можливо отримати при розробці власного об’єктно-орієнтовного редактору JSON формату, для розробки системи обрано саме цей варіант.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі були розглянуті технології, що будуть використані для написання системи – об'єкта розробки (елементів інтерфейсу користувача в середовищі операційної системи iOS), а також було зроблено вибір технологій реалізації програмних компонентів і сервісів, необхідних для задоволення потреб системи. Також було обґрунтовано, навіщо ці технології використовуватимуться та які переваги вони надаватимуть при їх використанні.

Спочатку було описано вимоги до системи і аргументовано розділення сфер засобів реалізації, необхідних для втілення умов задачі.

В якості мови програмування, яка буде використана для створення програмних компонентів системи, було обрано мову програмування Swift. Аргументує даний вибір простота у використанні, надійність, і високий рівень взаємодії із системними компонентами.

В якості бібліотеки для створення елементів інтерфейсу користувача у поєднанні з обраною мовою програмування, було розглянуто два основні варіанти реалізації: UIKit і SwiftUI. Після аналізу розглянутих рішень, вибір було зроблено в сторону фреймворку SwiftUI.

Для реалізації збереження і передачі даних конфігурації елементів UI було обрано тип даних *Struct* із кодуванням і розкодуванням у формат JSON. Така комбінація часто використовується розробниками ПО і гарантує надійність в процесі життєвого циклу додатку. Водночас, передача і збереження конфігурації у форматі JSON має зрозумілий вигляд для користувача системи. Також, було описано теоретичний вигляд даних, який буде використовуватись у системі, який зумовлює зрозумілий користувачу і системі варіант збереження даних.

В якості сервісу віддаленої конфігурації було обрано сервіс Firebase Remote Config через відсутність значної конкуренції на ринку. Реалізація власного сервісу не є пріоритетом поставленої задачі і має значні часові затрати.

Для реалізації редагування конфігурації було розглянуті можливі готові рішення, а також, проаналізовано необхідність створення власного інструментарію. Для збільшення надійності системи було зроблено вибір в сторону власної реалізації редактору JSON даних з об'єктним підходом.

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

Відповідно до досліджених матеріалів та обраного способу реалізації ми можемо перейти до фази розробки програмного продукту. Для того, щоб створити гнучку та настроювану систему віддаленого налаштування інтерфейсів користувача, необхідно описати основні частини розробленої системи, функціональні блоки програмних компонентів та схему їх взаємодії.

Виходячи із аналізу вимог до системи і розглянутих варіантів засобів реалізації системи, варто розбити розробку системи на 4 основні компоненти:

1. Розробка програмних компонентів бібліотеки елементів інтерфейсу користувача.
2. Розробка програмних компонентів редактору конфігурацій.
3. Інтеграція бібліотеки елементів інтерфейсу користувача.
4. Налаштування сервісу віддаленої конфігурації.

Винесення програмних компонентів елементів інтерфейсу користувача у окрему бібліотеку зменшує складності при інтеграції розробленої системи до ймовірної програми і дозволяє перевикористовувати написаний код. Окрім цього даний підхід дозволяє підключити систему до редактору конфігурації для миттєвого перегляду налаштованого інтерфейсу користувача.

3.1 Розробка програмних компонентів бібліотеки елементів інтерфейсу користувача

Для вирішення задачі проекту було розроблено бібліотеку елементів інтерфейсу користувача – DecodableUI. Форма групування коду у бібліотеку дозволяє підключати і перевикористовувати ділянки коду між різними

проектами. Підключення бібліотеки DecodableUI виконується за допомогою менеджера залежностей SPM.

Далі буде описано основні компоненти бібліотеки елементів інтерфейсу користувача і сервісу для розкодування даних конфігурації у відповідні елементи UI.

3.1.1 Структура DecodableViewConfiguration

Структура *DecodableViewConfiguration* представлє собою трансформовані у мову Swift налаштування інтерфейсів користувача. Дана структура зберігає у собі інформацію про назву елемента інтерфейсу і правила для розкодування внутрішніх параметрів елементу інтерфейсу.

DecodableViewConfiguration має 2 поля зберігаємих даних: *type*: *String* та *nestedDecoder*: *Optional<Decoder>*, а також функцію ініціалізації.

Поле *type* є константною змінною типу *String* (рядок). Ця змінна зберігає дані про назву типу елементу інтерфейсу користувача. Згідно із цим значенням, система розуміє який тип треба буде ініціалізувати при відповідному запиті.

Поле *nestedDecoder* є константною змінною типу *Optional<Decoder>*. Ця змінна зберігає опціональний екземпляр декодеру внутрішніх параментрів елементу інтерфейсу, тобто внутрішній контейнер даних декодування. Такий підхід дає змогу використовувати *DecodableViewConfiguration* для збереження даних конфігурації будь-яких елементів користувача, не вдаючись у подробиці імплементації цих елементів і параметрів які вони приймають. Структури, які можуть ініціалізуватись із *DecodableViewConfiguration* самі визначають правила для розкодування параметрів, а саме: їх кількість, типи, значення тощо.

Функція *init(from decoder: Decoder) throws* означає конформацію *DecodableViewConfiguration* протоколу *Decodable*, тобто цю ініціалізувати екземпляри цієї структури можна із декодеру даних, тобто напряму із JSON-форматованого запису конфігурації.

3.1.2 Протокол DecodableView

Протокол *DecodableView* означає набір правил до структур елементів інтерфейсу користувача, яким останні повинні підпорядковуватись для роботи системи. *DecodableView* надає рівень абстракції для роботи із конфігурованими елементами інтерфейсу користувача.

DecodableView зберігає одне обчислювальне поле даних – *anyView: AnyView*. Протокол *AnyView* – частина фреймворку SwiftUI, яка означає належність структури до елементів інтерфейсу, здатних до відмальовування на екрані користувача і взаємодії із ним. Поле *anyView* буде вираховуватись кожною структурою, яка підпорядковується протоколу *DecodableView* самостійно.

Функція ініціалізації структур, які підпорядковуються *DecodableView* приймає два параметри: *decoder: Optional<Decoder>*, та *viewResolver: DecodableViewResolver*. Останній буде розглянуто у наступному пункті (п. 3.1.3). Параметр *decoder: Optional<Decoder>* являє собою внутрішній контейнер параметрів елементу інтерфейсу користувача, як *nestedDecoder* у структурі *DecodableViewConfiguration* (див. п. 3.1.1).

3.1.3 Протокол DecodableViewResolver

Протокол *DecodableViewResolver* являє собою абстрактний тип, який здатний до створення інстансів елементів інтерфейсу користувача на етапі рантайму додатку.

В правилах протоколу лежить метод *resolve(from configuration: DecodableViewConfiguration) -> DecodableView?*. Даний метод означає можливість класів і структур перетворювати інстанси налаштувань типу *DecodableViewConfiguration* у елементи інтерфейсу *DecodableView.*

Результуючий тип функції є опціональним, оскільки не кожну конфігурацію можливо перетворити у елементи інтерфейсу.

Екземпляр типу, підпорядкованому протоколу *DecodableViewResolver*, необхідний для передачі як залежність, оскільки елементи інтерфейсу можуть містити внутрішні елементи, які потребують резолюції. Було вирішено зробити це саме підходом DI, оскільки для різних компонентів програми можуть потребуватись різні конфігурації типів *DecodableViewResolver*, і підхід з використанням архітектурного паттерну Singleton не є належним рішенням через свою сингулярність.

3.1.4 Клас DecodableViewsService

Клас *DecodableViewsService* являє собою головну точку входу нашої бібліотеки. Даний клас являє собою регістр типів елементів інтерфейсу користувача – *DecodableView*.

DecodableViewsService має поле *viewsRegistry: [String: DecodableView.Type]*, тобто словник, де ключем є назва типу елементу, а значенням – його фактичний тип. Оскільки Swift є статично-типізованою мовою програмування, даний підхід є елегантним підходом для створення інстансів фактичних типів *DecodableView* із екземплярів *DecodableViewConfiguration*.

Користувач бібліотеки передає необхідні типи, які підпорядковуються протоколу *DecodableView* на реєстрацію в регістрі за допомогою методу *register(_ viewTypes: [String: DecodableView.Type])*.

DecodableViewsService підпорядковується протоколу *DecodableViewResolver* завдяки імплементації методу *resolve(from configuration: DecodableViewConfiguration) -> DecodableView?*.

```
public func resolve(
    from configuration: DecodableViewConfiguration
) -> DecodableView? {
    guard let type = viewsRegistry[configuration.type],
          let view = type.init(from: configuration.nestedDecoder, viewResolver: self) else {
              return nil
          }

    return view
}
```

Рисунок 3.1 – реалізація методу резолюції елементу інтерфейсу із його конфігурації

3.1.5 Протокол DecodableViewModifier

Оскільки елементи інтерфейсу мають деякі загальні параметри конфігурації (ширина і висота на екрані користувача, скруглення, фон тощо), було вирішено створити протокол *DecodableViewModifier*, серед правил якого є лише можливість створення інстансу типу із декодеру: *init?(from decoder: Decoder?)*.

DecodableViewModifier наслідується від протоколу *ViewModifier* із фреймворку SwiftUI. Це значить, що тип *DecodableViewModifier* мусить визначити метод модифікації елементу інтерфейсу, тобто у декларативному вигляді налаштувати його вигляд чи поведінку.

Користувач бібліотеки може створювати свої типи *DecodableViewModifier* для зменшення коду при створенні власних елементів інтерфейсу користувача якщо вони мають спільні параметри.

3.1.6 Структура DefaultViewModifier

DefaultViewModifier це фактичний тип, підпорядкований протоколу *DecodableViewModifier*.

Даний *ViewModifier* автоматично витягує параметри *background (фон), cornerRadius* (скруглення)*, padding* (відступи)*, width* (ширина) та *height* (висота) із конфігурації *DecodableViewConfiguration*, і додає ці налаштування до обраного елементу конфігурації. При відсутності якогось із цих параметрів в налаштуваннях елементу, встановлюються системні значення цих налаштувань.

3.1.7 Стандартні елементи інтерфейсу користувача

Бібліотека доставляє деякі загальні види елементів інтерфейсу за замовчуванням. До таких елементів належать *LabelView* (текст), *ImageView* (зображення), та *StackView* (группа елементів). Кожен із цих типів підпорядковується протоколу *DecodableView*, та є генерік типом. В якості підлежного типу до ініціалізації екземплярів даних типів передається тип *DecodableViewModifier*.

Спосіб ініціалізації цих компонентів наслідується від батьківського класу *DecodableView*.

Деталі імплементації стандартних елементів інтерфейсу користувача будуть розглянуті у пунктах 3.1.8-3.1.10.

3.1.8 Структура LabelView

LabelView являє собою елемент тексту на екрані користувача. В якості параметрів у конфігурації прописується текст елементу, колір і розмір шрифта.

Параметри, які приймає *LabelView* для ініціалізації:

1. *text: String* – фактичний текст, який необхідно відобразити на екрані користувача.

2. *fontSize: CGFloat* – розмір шрифту тексту. Одиниця вимірювання: поінти [11]. За замовчуванням, тобто при відсутності значень, встановлюється системний розмір тексту.
3. *lineLimit: Int* – максимальна кількість ліній тексту. За замовчуванням, або при відсутності даних, обмежень по кількості ліній не встановлюється.
4. *fontColor: Color* – колір тексту. За замовчуванням, або при відсутності даних, встановлюється системний колір тексту.
5. *viewModifier*: *DecodableViewModifier* – модифікатор загальних налаштувань (див. п. 3.1.5).

Ініціалізація із формату JSON виконана у розширенні структури для підпорядкування протоколу *DecodableView*.

```
public init?(from decoder: Decoder?, viewResolver: DecodableViewResolver) {
    guard let container = try? decoder?.container(keyedBy: CodingKeys.self),
          let text = try? container.decode(String.self, forKey: .text) else {
        return nil
    }

    let fontSize = try? container.decode(CGFloat.self, forKey: .fontSize)
    let lineLimit = try? container.decode(Int.self, forKey: .lineLimit)
    let fontColor = try? container.decodeColorFromHexString(forKey: .fontColor)

    let viewModifier = Modifier(from: decoder)

    self.init(
        text: text,
        fontSize: fontSize,
        lineLimit: lineLimit,
        fontColor: fontColor,
        viewModifier: viewModifier
    )
}
```

Рисунок 3.2 – ініціалізація *LabelView* із декодеру JSON формату.

Таким чином, можливий варіант конфігурації *LabelView* у JSON форматі зображено на рисунку 3.2.

```
{
  "type": "Label",
  "parameters": {
    "text": "Some Text",
    "fontSize": 44,
    "fontColor": "#a1a1a1"
  }
}
```

Рисунок 3.2 – приклад конфігурації елементу *LabelView* у форматі JSON.

3.1.9 Структура ImageView

ImageView являє собою зображення на екрані користувача, яке може бути завантажене через шлях до зображення або через системне ім’я зображення, які беруться із конфігурації елементу.

Параметри, які приймає *ImageView* для ініціалізації:

1. *systemName: String* – назва системного зображення із колекції SF Symbols[12].
2. *url: Url?* – посилання на зображення. За відстунтості параметру, у контейнер зображення встановлюється системне.
3. *viewModifier*: *DecodableViewModifier* – модифікатор загальних налаштувань (див. п. 3.1.5).

Ініціалізація із формату JSON виконана у розширенні структури для підпорядкування протоколу *DecodableView*.

```swift
public init?(from decoder: Decoder?, viewResolver: DecodableViewResolver) {
    let container = try? decoder?.container(keyedBy: CodingKeys.self)

    let url = try? container?.decode(URL.self, forKey: .url)
    let systemName = try? container?.decode(String.self, forKey: .systemName)

    let viewModifier = Modifier(from: decoder)

    self.init(
        url: url,
        systemName: systemName,
        viewModifier: viewModifier
    )
}
```

Рисунок 3.3 – ініціалізація *ImageView* із декодеру JSON формату.

Таким чином, можливий варіант конфігурації *ImageView* у JSON форматі зображено на рисунку 3.4.

```json
{
  "type": "Image",
  "parameters": {
    "url": "https://some-url.com",
    "systemName": "photo"
  }
}
```

Рисунок 3.4 – приклад конфігурації елементу *ImageView* у форматі JSON.

3.1.10 Структура StackView

StackView являє собою групу елементів на екрані користувача. В параметрах конфігурації *StackView* прописується напрямок розміщення елементів (горизонтальний чи вертикальний), вирівнювання відносно центру напрямку, відстань між внутрішніми елементами, та конфігурації внутрішніх елементів у масиві. Таким чином, *StackView* дозволяє компонувати різні елементи інтерфейсу будь-якої складності. Для оптимізації елементу у більш вузьконапрямлених задачах, користувач бібліотеки має змогу створювати

власні елементи інтерфейсу, які більше підходять для вирішення поставлених йому задачам (наприклад, елемент список чи таблиця).

Параметри, які приймає *StackView* для ініціалізації:

1. *direction: Direction (enum)* – напрямок розміщення елементів інтерфейсу: горизонтальний (.horizontal) або вертикальний (.vertical).
2. *spacing*: *CGFloat* – відстань між внутрішніми елементами. Одиниця вимірювання: поінти. За замовчуванням, чи за відсутності даних, встановлюється системне значення.
3. *alignment*: *Alignment* – вирівнювання відносно центру напрямку. Для горизонтального напрямку: лівий край (.leading), правий край (.trailing), чи центр (.center). Для вертикального напрямку: верхня границя (.top), нижня границя (.bottom), чи центр (.center). За замовчуванням, чи за відсутності даних, встановлюється вирівнювання по центру напрямку.
4. *elements*: *[AnyView]* – масив внутрішніх елементів.
5. *viewModifier*: *DecodableViewModifier* – модифікатор загальних налаштувань (див. п. 3.1.5).

```
public init?(from decoder: Decoder?, viewResolver: DecodableViewResolver) {
    guard let container = try? decoder?.container(keyedBy: CodingKeys.self),
          let direction = try? container.decode(Direction.self, forKey: .direction),
          let configurations = try? container.decode([DecodableViewConfiguration].self, forKey: .elements) else {
              return nil
          }

    let elements = configurations.compactMap { configuration in
        viewResolver.resolve(from: configuration)?.anyView
    }
    let alignment = try? container.decode(Alignment.self, forKey: .alignment)
    let spacing = try? container.decode(CGFloat.self, forKey: .spacing)

    let viewModifier = Modifier(from: decoder)

    self.init(
        direction: direction,
        spacing: spacing,
        alignment: alignment,
        viewModifier: viewModifier,
        elements: elements
    )
}
```

Рисунок 3.5 – ініціалізація *StackView* із декодеру JSON формату.

Ініціалізація із формату JSON виконана у розширенні структури для підпорядкування протоколу *DecodableView*. Тут задаються необхідні для елементу параметри конфігурації із даних налаштувань. Для внутрішніх елементів використовується *DecodableViewResolver*, який передається по ієрархії елементів методом DI.

Таким чином, можливий варіант конфігурації *StackView* у JSON форматі зображено на рисунку 3.6. На даному рисунку відображена конфігурація вертикальної групи елементів – зображення і тексту.

```
{
  "type": "Stack",
  "parameters": {
    "direction": "vertical",
    "elements": [
      {
        "type": "Image",
        "parameters": {
          "systemName": "photo"
        }
      },
      {
        "type": "Label",
        "parameters": {
          "text": "Some text"
        }
      }
    ]
  }
}
```

Рисунок 3.6 – приклад конфігурації елементу *StackView* у форматі JSON

3.2 Розробка програмних компонентів редактору конфігурацій.

Для зручного редагування конфігураційних даних формату JSON було розроблено об’єктний редактор для операційної системи MacOS із використанням фреймворку SwiftUI. Даний редактор дозволяє без помилок

редагувати конфігурацію елементів інтерфейсу користувача, і оглядати результат налаштувань у режимі реального часу.

Змістовно розробку редактору конфігурацій можна розбити на три основні компоненти:

1. Бізнес-логіка представлення JSON формату у структури даних, які можна представити у мові програмування Swift.
2. Відображення конфігурації у зрозумілому для користувача вигляді і надання йому інтерфейсів взаємодії із ними.
3. Перетворення у режимі реального часу сконфігурованих даних у елементи інтерфейсу користувача за допомогою розробленої бібліотеки *DecodableUI*.

Далі буде детально описано основні компоненти редактору конфігурацій інтерфейсу користувача.

3.2.1 Структура JSONRow

Реалізацію представлення даних JSON формату у мові Swift було вирішено зробити за допомогою перераховуваного типу *JSONRow*. Структура *JSONRow* представляє собою рядок даних у об’єкті формату JSON, тобто ключ і значення. *JSONRow* має 2 поля: *key: String?* та *value: JSONValue* (див. пункт 3.2.2). Поле *key* є опціональним, оскільки в межах даного рішення було вирішено представити елементи масиву саме як *JSONRow* для полегшення обчислень і перемальовування інтерфейсу.

JSONRow підпорядковується протоколам *Identifiable* та *Hashable*. Тобто ми можемо отримати інформацію про індивідуальність екземплярів структури *JSONRow* і інформацію про зміну значення даної структури, що необхідно при створенні інтерфейсу користувача для взяємодії с цими даними.

JSONRow має низку декораторів, які виступають для вирішення проблем і покращення стилю коду:

1. *imageSystemName: String* – обчислюєма змінна, яка повертає назву зображення, яке відповідає поточному типу рядка.
2. *jsonString: String* – обчислюєма змінна, яка повертає значення рядка у форматі JSON.
3. *nestedRows: [JSONRow]* – обчислюєма змінна, яка повертає усі вкладені рядки даного, якщо даний рядок має тип масив або об’єкт. В інших випадках *nestedRows* повертає порожній масив.
4. *mutating func removeRow(with id: UUID) -> JSONRow?* – інтерфейс для видалення рядка із поданим ідентифікатором рекурсивним методом. Якщо у даного рядка є вкладені рядки, для кожного з них викликається даний метод. При знаходжені рядка із поданим унікальним ідентифікатором, він видаляється з вкладених рядків у батьківського.
5. *mutating func create(after id: UUID) -> JSONRow?* – інтерфейс для створення нового рядка. Спочатку рекурсивним шляхом знаходиться рядок із поданим унікальним ідентифікатором. Якщо знайдений рядок може містити вкладені (тобто цей рядок має тип значення масив або об’єкт), до його вкладених додається новий із порожнім типом, який користувач зможе змінити далі за допомогою відповідних методів. Якщо знайдений рядок не може містити вкладених рядків, то новий рядок створюється після даного у батьківському рядку.
6. *previousWideRow(for id: UUID) -> JSONRow?* – метод для знаходження попереднього рядку у ієрархії рекурсивним методом. Цей інтерфейс використовується для встановлення фокусу програми на попередній рядок після видалення рядку, який знаходиться у фокусі.

3.2.2 enum JSONValue

JSONValue представляє собою можливість змінної перебувати у стані певного типу аргументу значення JSON формату. *JSONValue* є асоційованим

перераховуваним типом. Кожному стану значення відповідають різні типи асоційованих значень

Розглянемо можливі типи даних, які представлені у JSON форматі і наведемо їх представлення у нашому рішенні:

1. Об'єкт: *case object* із асоційованим типом *[JSONRow]*. Таким чином рядок, тип значення якого JSONValue.object може матии вкладені рядки.
2. Масив: *case array* із асоційованим типом *[JSONRow]*. Тут варто пам'ятати, що вкладені елементи *JSONRow* не повинні мати ключа. Інтерфейс програми не дозволить користувачу створити елементи масиву із ключами.
3. Рядок: *case string* із асоційованим типом *String.* Даний тип не може мати вкладених елементів.
4. Число: *case number* із асоційованим типом *Int*. Даний тип не може мати вкладених елементів.
5. Булеве значення: *case bool* із асоційованим типом *Bool.* Даний тив не може мати вкладених елементів.

JSONValue має низку декораторів, які виступають для вирішення проблем і покращення стилю коду:

1. *jsonString: String* – обчислювальна змінна, яка повертає значення *JSONValue* у форматі JSON.
2. *objectValues*: *[JSONRow]* – обчислювальна змінна, яка повертає асоційоване значення у випадку, якщо змінна є типом *JSONValue.object*.
3. *arrayValues*: *[JSONRow]* – обчислювальна змінна, яка повертає асоційоване значення у випадку, якщо змінна є типом *JSONValue.array*.
4. *stringValue*: *String* – обчислювальна змінна, яка повертає асоційоване значення у випадку, якщо змінна є типом *JSONValue.string*.

5. *numberValue*: *Int* – обчислювальна змінна, яка повертає асоційоване значення у випадку, якщо змінна є типом *JSONValue.number*.
6. *boolValue*: *Bool* – обчислювальна змінна, яка повертає асоційоване значення у випадку, якщо змінна є типом *JSONValue.bool*.
7. метод *hash(into hasher: inout Hasher)* – метод, який повертає хеш значення екземпляру *JSONValue* шляхо поєднання значень його можливих асоційованих значень. Цей метод допомагає дізнатись про зміни типу екземпляру і його асоційованих значень.
8. *valueDescription: String* – обчислювальна змінна, яка повертає у зрозумілому для користувача вигляді значення екземпляру *JSONValue.*
9. *typeDescription: String* – обчислювальна змінна, яка повертає у зрозумілому для користувача вигляді тип екземпляру *JSONValue.*

3.2.3 Реалізація інтерфесу користувача

Реалізація інтерфейсу користувача здійснена із використанням елементів інтерфейсу фреймфворку SwiftUI. Весь інтерфейс користувача знаходиться у структурі *JSONEditor*, яка декларативним чином об’єднує всі розроблені елементи інтерфейсу користувача для вирішення поставленої проблеми.

Візуально розроблений інтерфейс користувача можна розділити на наступні компоненти:

1. Навігація. Навігація представляє собою ієрархію об’єкту JSON у зрозумілому для користувача вигляді. Реалізація ієрархії здійснюється рекурсивним шляхом у структурі *JSONHierarchy*. Кожний елемент ієрархії є структурою *JSONHierarchyRow*, який за допомогою розроблених вище декораторів перетворює значення типу *JSONRow* у зрозумілий для користувача вигляд. Користувач може створювати і видаляти елементи ієрархії за допомою контекстного меню,

реалізованого вбудованими елементами фреймворку SwiftUI. Далі вибрана користувачем дія викликає відповідні інтерфейси обраного рядка *JSONRow – create(after id: UUID)* або *removeRow(with id: UUID)*. При виборі елементу у ієрархії на основному екрані з'являється інтерфейс для редагування обраного елементу *JSONRow*.

2. Інтерфейс редагування обраного елементу. Даний інтерфейс реалізований завдяки структурі *JSONRowEditor*. В залежності від типу значення рядку *JSONRowEditor* відображає коректну форму взаємодії із значенням.
3. Інтерфейс показу сконфігурованого інтерфесу. Даний інтерфейс реалізований за допомогою компонентів розробленої бібліотеки *DecodableUI*.

3.2.4 Інтеграція розробленої бібліотеки DecodableUI у редактор конфігурацій

Розроблений інтерфейс дозволяє модифікувати корінь JSON об'єкту типу *JSONRow*. Для перетворення *JSONRow* у рядок формату JSON в режимі реального було створено реактувну модель *JSONModel*, яка містить інформацію про корінь об'єку та надає інтерфеси для його модифікації. *JSONModel* має наступні поля і методи:

1. *@Published var selectedId: UUID?* – зберігає дані про обраний користувачем рядок *JSONRow*. Дана інформація дозволяє зрозуміти над яким рядком користувач виконює дії і який рядок треба відкрити в центрі екрану для модифікації значень.
2. *@Published var rootObject: JSONRow* – зберігає корінь об'єкту JSON.
3. *private var cancellables*: *Set<AnyCancellable>* - колектор підписок і їх утилізацій.

4. *var jsonPublisher*: *AnyPublisher<String, Never>* – паблішер до перетвореного *JSONRow* у рядок. Дане поле дає інтерфейс для підписки інших об’єктів до змін конфігурації формату JSON.
5. *func isRootObject(_ row: JSONRow) -> Bool* – інтерфейс який надає інформацію, чи вхідний рядок *JSONRow* є корінним. Це робиться для відображення користувачу можливості для видалення рядку. Якщо обраний рядок є коренем об’єкту, його модифікація не є можливою.
6. *func create(after id: UUID)*, *func delete(with id: UUID)* та *func deleteSelected()* – інтерфейси для створення та видалення рядків.
7. *Func setSelected(id: UUID?)* – інтерфейс для зміни фокусу на обраний рядок.

На рис. 3.7 представлено діаграму *JSONModel*.

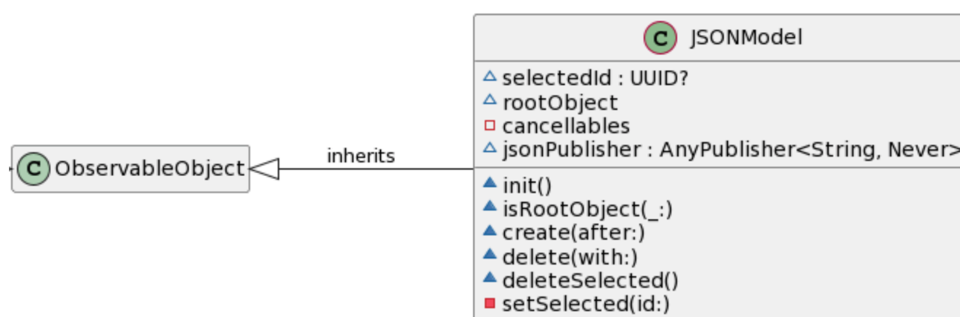


Рисунок 3.7 – UML діаграма класу *JSONModel*

Для відображення рядку формату JSON у елементи UI в режимі реального часу було створено реактивну модель *DecodableUIProvider*. Дана модель має залежності на розроблену бібліотеку *DecodableUI* і дозволяє в режимі реального часу отримати структуру типу *AnyView* із рядку конфігурації формату JSON. *DecodableUIProvider* підписується на зміни поля *jsonPublisher* моделі *JSONModel* і перетворює конфігурацію у структуру *AnyView* за допомогою сервісів *DecodableViewsService* із розробленої бібліотеки *DecodableUI*. *DecodableViewsService* має наступні поля і методи:

1. *private let viewsService: DecodableViewsService* – екземпляр класу *DecodableViewsService* із розробленої бібліотеки *DecodableUI*.

2. *@Published var view: AnyView?* – реактивне представлення конфігурації JSON формату у елемент інтерфейсу. За відсутності конфігурації, або під час перетворення конфігурації у елементи, ця змінна набуває значеня структури, яка відображає завантаження елементу. Якщо конфігурацію неможливо перетворити у елемент інтерфейсу, дана змінна набуває значення структури, яка відображає користувачу інформацію про помилку.
3. *@Published var configuration: DecodableViewConfiguration?* – перетворений рядок формату JSON у прийнятий розробленою бібліотекою *DecodableUI* формат *DecodableViewConfiguration.* За неможливості перетворення рядку формату JSON у формат *DecodableViewConfiguration*, або за відсутності рядку формату JSON, ця змінна набуває значення nil, тобто відсутність значення.
4. *private var cancellables*: *Set<AnyCancellable>* - колектор підписок і їх утилізацій.
5. *init(viewTypes: [String: DecodableView.Type])* – метод ініціалізації екземпляру *DecodableUIProvider* із наданим йому словником можливих елементів інтерфейсу.
6. *var jsonSubject = PassthroughSubject<String, Never>* - змінна перетворюваного на даний момент рядку формату JSON. Задається об'єктом-медіатором між класами *JSONModel* і *DecodableUIProvider*..
7. *private var configurationPublisher: AnyPublisher<DecodableViewConfiguration?, Never>* - паблішер конфігурації типу *DecodableViewConfiguration*. Ця змінна підписується на зміни *jsonSubject* і трансформує рядок конфігурації формату JSON у формат *DecodableViewConfiguration* в режимі реального часу.

На рис. 3.8 представлено діаграму *DecodableUIProvider*.

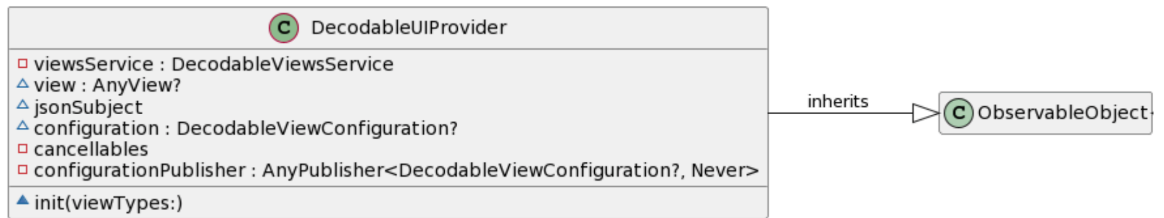


Рисунок 3.8 – UML діаграма класу *DecodableUIProvider*

3.3 Налаштування сервісу віддаленої конфігурації

Налаштування сервісу віддаленої конфігурації Firebase Remote Config детально описане у документації сервісу [13].

Першим кроком є створення проекту у консолі розробника Firebase Console. Після введеня необхідних налаштувань проекту, таких як використання A/B тестів, аналітики, назви проекту тощо, відбувається інтеграція Firebase проекту у клієнтський додаток.

Інтеграція проекту Firebase відбувається шляхом реєстрації ідентифікатора додатку, завантаженням конфігураційних файлів проекту *GoogleService-info.plist* в середовище XCode, завантаженням SDK Firebase [14], та ініціалізацією проекту в коді додатку.

Додавання SDK Firebase у проект додатку виконується шляхом додання залежності через менеджер залежностей SPM.

```swift
import SwiftUI
import FirebaseCore

class AppDelegate: NSObject, UIApplicationDelegate {
    func application(
        _ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]? = nil
    ) -> Bool {
        FirebaseApp.configure()

        return true
    }
}

@main
struct ThesisApp: App {
    // register app delegate for Firebase setup
    @UIApplicationDelegateAdaptor(AppDelegate.self) var delegate

    var body: some Scene {
        WindowGroup {
            NavigationView {
                ContentView()
            }
        }
    }
}
```

Ініціалізація інструментів SDK Firebase виконується завдяки виклику відповідного методу бібліотеки FirebaseCore.

Рисунок 3.9 – Ініціалізація інструментів SDK Firebase

Для створення віддаленої конфігурації у сервісі Firebase Remote Config необхідно натиснути на кнопку «Add parameter» в консолі, обрати тип JSON, ввести назву параметру, та заповнити налаштування елементу інтерфейсу у відповідному полі.

Parameter name (key) ⓘ

userInterface

Data type

{} JSON

Description

Optional

Default value

Use in-app default

{"type":"Stack","parameters":{"elements":[{"type":"Label","parameters":{"tex

Add new

Cancel

Save

Рисунок 3.10 – додавання налаштування елементу інтерфейсу користувача в консолі Firebase Remote Config

3.4 Інтеграція розробленої бібліотеки DecodableUI у клієнтські додатки

Інтеграція розробленої бібліотеки *DecodableUI* у клієнтські додатки має вигляд, подібний до інтеграції, розглянутої у п. 3.2.4, із деякими відмінностями.

Налаштування елементів інтерфейсу користувача надається із сервісу віддаленої конфігурації елементів UI із подальшим перетворенням рядків формату JSON у елементи розробленої бібліотеки *DecodableUI*.

Після успішної ініціалізації SDK Firebase, розглянутої у п. 3.3, треба налаштувати інтерфейс отримання віддаленої конфігурації у клієнтському додатку. Для цього було створено низку протоколів і класів, необхідних для правільної інтеграції залежності у проект.

Протокол *RemoteConfigurationProvider* надає абстрактний інтерфейс для завантаження віддаленої конфігурації і є точкою входу для сервісів віддаленої конфігурації у додатку. *RemoteConfigurationProvider* має два обов'язкові методи:

1. *fetchConfig(completion: @escaping (Error?) -> Void)* – функція вищого порядку, яка викликається за завантаження конфігурації і передає інформацію про результат завантаження у замикання.
2. *getValue(forKey key: String) -> Result<String, Error>* - метод для отримання значень конфігурації по ключу. Може повертати результуючий рядок, ябо помилку, якщо даного запису не існує.

Клас *FirebaseRemoteConfigurationDownloader* є фактичним класом який обертає логіку взаємодії із SDK Firebase для роботи із віддаленими налаштуваннями. В ініціалізації класу *FirebaseRemoteConfigurationDownloader* виконуються необхідні налаштування SDK Firebase, які стосуються сервісів віддаленого налаштування. Клас *FirebaseRemoteConfigurationDownloader* підпорядковується протоколу *RemoteConfigurationProvider*.

```swift
extension FirebaseRemoteConfigurationDownloader: RemoteConfigurationProvider {

    func fetchConfig(completion: @escaping (Error?) -> Void) {
        remoteConfig.fetch { [remoteConfig] (status, error) -> Void in
            guard status == .success else {
                completion(RemoteConfigurationError.loadingError)
                return
            }
            remoteConfig.activate()
            completion(nil)
        }
    }

    func getValue(forKey key: String) -> Result<String, Error> {
        let value = remoteConfig.configValue(forKey: key).dataValue
        guard let string = String(data: value, encoding: .utf8) else {
            return .failure(RemoteConfigurationError.valueError)
        }
        return .success(string)
    }

}
```

Рисунок 3.11 – імплементація методів *RemoteConfig Provider* для класу *FirebaseRemoteConfigurationDownloader*

Для реактивного завантаження віддаленої конфігурації було розроблено клас *DecodableUIConfigurationService*, який взаємодіє із екземплярами *RemoteConfigProvider*. При ініціалізації класу, виконується завантаження віддаленої конфігурації і передача її до змінної *jsonSubject*, на яку підписується *DecodableUIProvider* (див. п. 3.2.4).

UML діаграма логіки інтеграції сервісу віддаленої конфігурації зображена на рисунку 3.12.

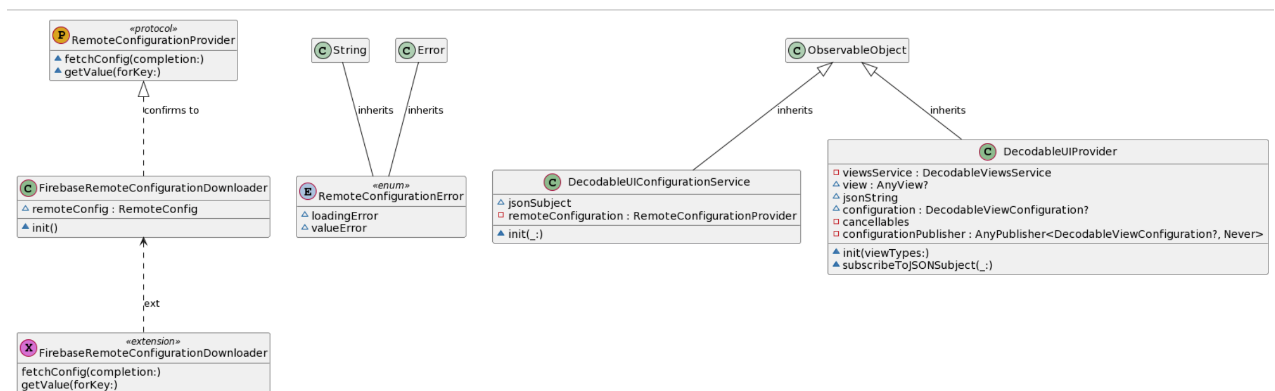


Рисунок 3.12 – UML діаграма інтеграції сервісу віддаленої конфігурації у клієнтський додаток в середовищі iOS

ВИСНОВОК ДО РОЗДІЛУ 3

В результаті проведеної роботи щодо розробки програмного забезпечення системи для реалізації віддаленого налаштування інтерфейсів користувача в клієнтських додатків в середовищі iOS можна виокремити наступне:

- Реалізовано раніше обгрунтовані (у Розділі 2) програмні компоненти системи.
- Наведено UML діаграми розроблених класів.
- Детально описано атрибути, методи та властивості необхідних класів і структур, необхідних для роботи системи.
- Були продемонстровані ключові фрагменти коду у вигляді скріншотів розроблюваної системи.
- Система складається із двох основних компонентів: бібліотека *DecodableUI* та редактор конфігурації елементів інтерфейсу користувача із відображенням налаштованих елементів UI у режимі реального часу.
- Було наведено інструкції щодо налаштування сервісу віддаленої коніфгурації і наведоно приклад інтеграції віддаленого сервісу конфігурації та розробленої бібліотеки *DecodableUI* у клієнтський додаток.

РОЗДІЛ 4. ВИКОРИСТАННЯ ТА РОЗШИРЕННЯ РОЗРОБЛЕНОЇ СИСТЕМИ

Базуючись на розробленій системі системи для реалізації віддаленого налаштування інтерфейсів користувача в клієнтських додатків в середовищі iOS метою даного розділу є показати результати її роботи і можливості щодо розширення функціоналу користувачем системи для втілення поставлених їм задач компонентами розробленої системи.

4.1 Огляд редактору конфігурацій

На рисунку 4.1 зображений розроблений програмний інтерфейс редактору конфігурацій елементів інтерфейсу користувача.

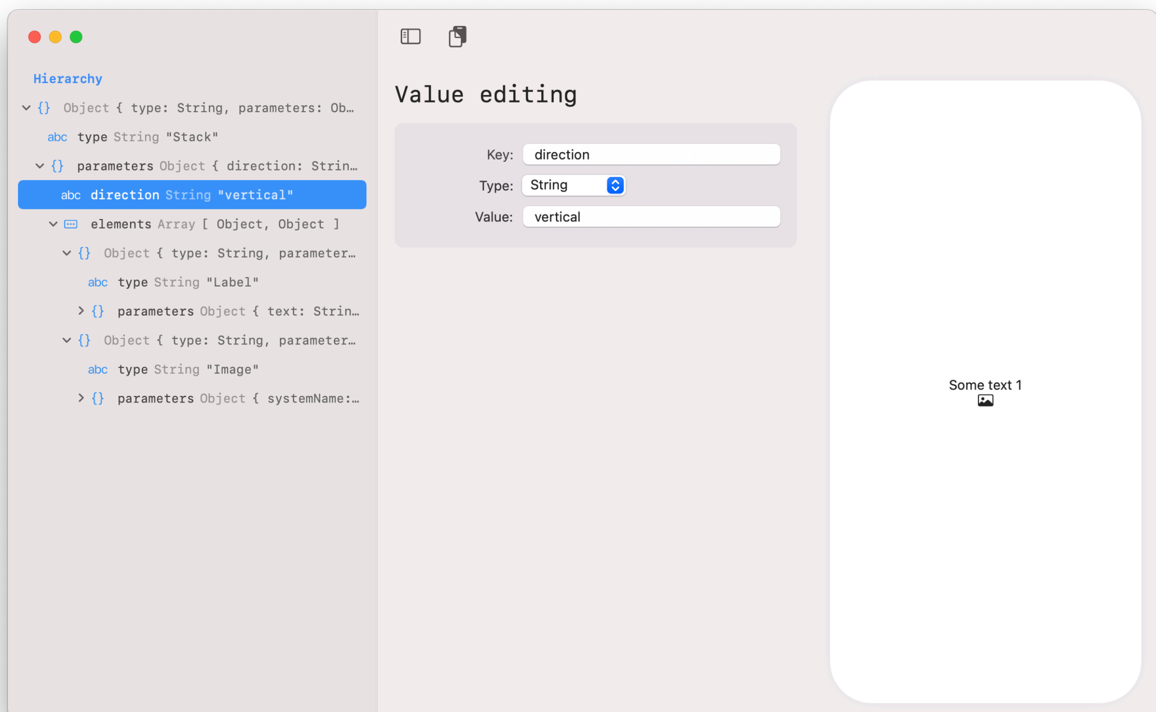


Рисунок 4.1 – програмний інтерфейс редактору конфігурацій

Зліва можна побачити ієрархію об’єкту формату JSON. Кожний рядок відповідає рядку запису у об’єкта. Для зручності використання, кожний рядок містить інформацію щодо його типу, скорочений опис значення і зображення, яке відповідає типу рядка. При натисканні на рядок правою кнопкою миші, з’являється контекстне меню для видалення радка, або створення нового в зоні обраного.

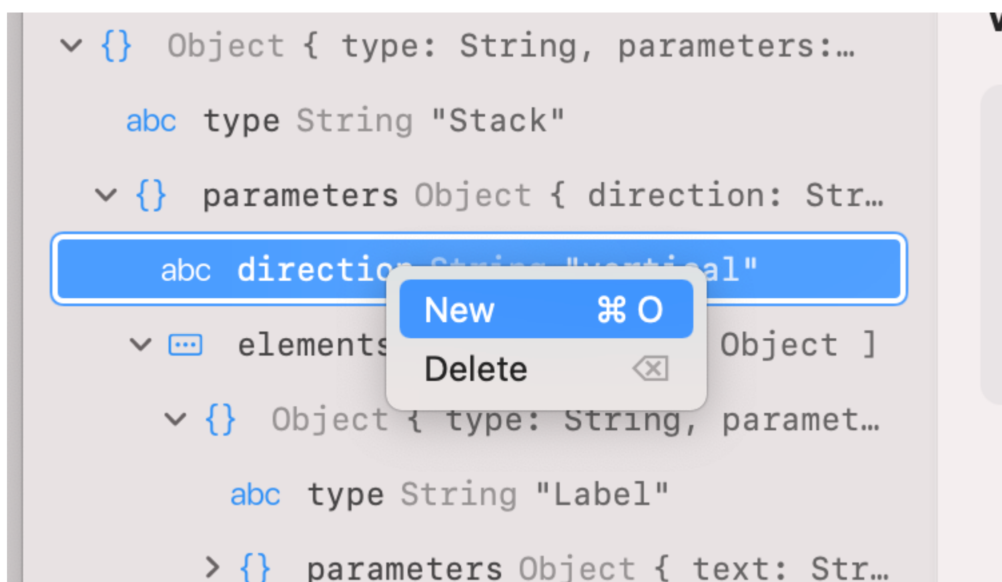


Рисунок 4.2 – контекстне меню створення і видалення рядку запису

Центральний сегмент програмного інтерфейсу, зображеного на рис. 4.1 відповідає за редагування обраного рядку запису. В ньому можна змінювати ключ рядка, а також тип і значення.

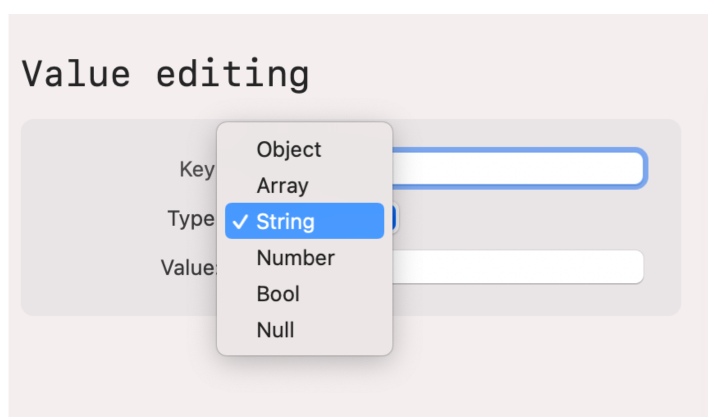


Рисунок 4.3 – контекстне меню вибору типу рядка

Права частина програмного інтерфейсу, зображеного на рис. 4.2 відображає сконфігуровані елементи інтерфейсу користувача. При зміні

конфігурації, налаштований елемент автоматично оновлюється у правій частині програмного інтерфейсу.

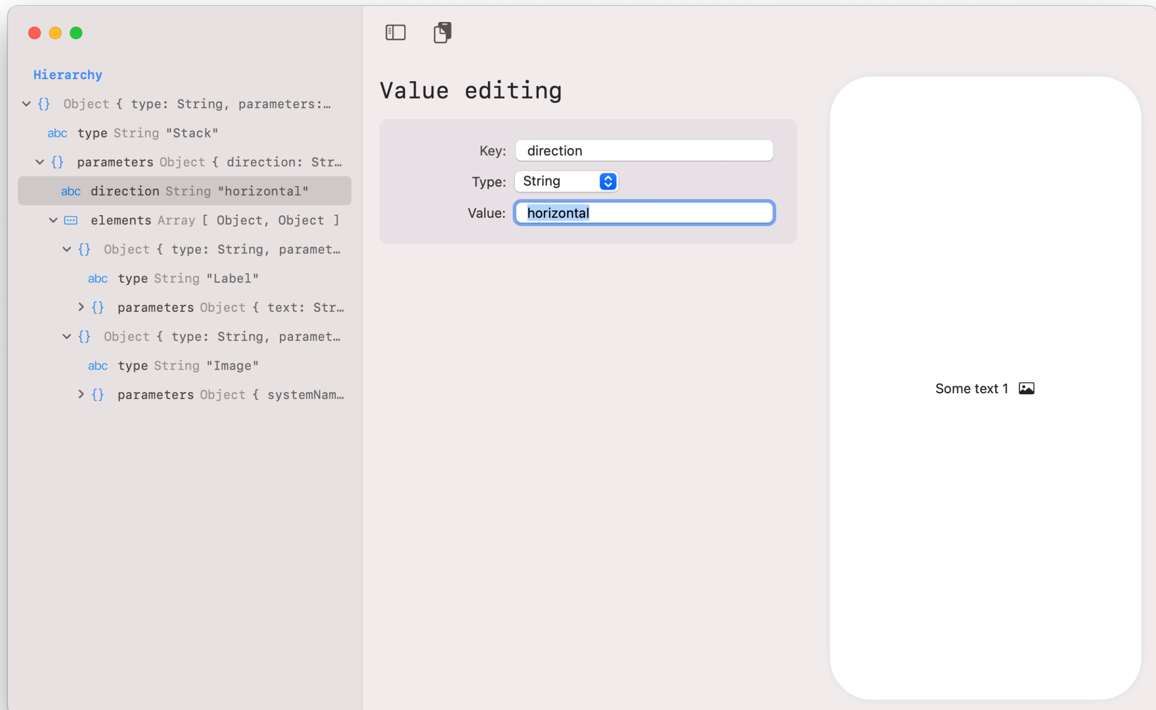


Рисунок 4.4 – демонстрація відображення змін конфігурації

Для отримання конфігурації у форматі рядку JSON формату, необхідно натиснути на відповідну кнопку у панелі інструментів. При настисканні на цю кнопку, рядок буде збережений у буфер обміну пристрою користувача, про що буде повідомлено користувачу редактора.

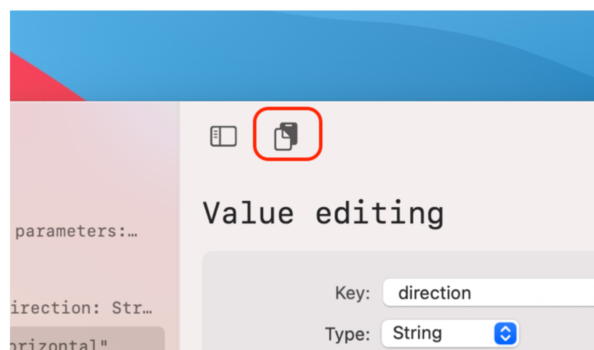


Рисунок 4.5 – кнопка копіювання конфігурації у буфер обміну пристрою

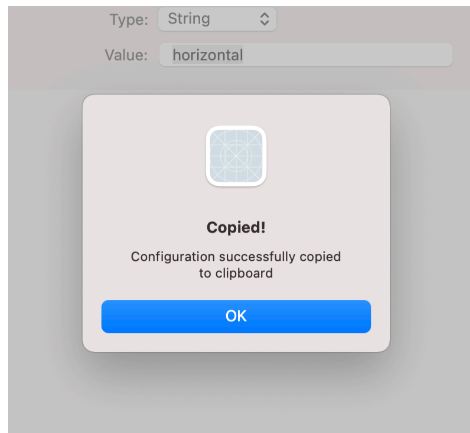


Рисунок 4.6 – повідомлення про успішне збереження конфігурації у буфер обміну

При копіюванні конфігурації у буфер обміну, рядок JSON форматується для зрозумілого користувачеві вигляду, як зображено на рисунку 4.7.


```
{
  "type" : "Stack",
  "parameters" : {
    "direction" : "horizontal",
    "elements" : [
      {
        "type" : "Label",
        "parameters" : {
          "text" : "Some text 1"
        }
      },
      {
        "type" : "Image",
        "parameters" : {
          "systemName" : "photo"
        }
      }
    ]
  }
}
```


Рисунок 4.7 – результат роботи редактору конфігурацій елементів інтерфейсу користувача у форматі JSON

Користувач системи може побачити і зрозуміти результати конфігурування елементів інтерфейсу завдяки розробленому редактору, а також, не має витрачати час на форматування конфігурації і синтаксис написання конфігурації у форматі JSON. Таким чином, можна вважати вимоги щодо оптимізації написання конфігурації і запобіганню помилок, які виникли внаслідок людського фактору, виконаними.

4.2 Використання конфігурації елементів інтерфейсу користувача

Для завантаження конфігурації, створеної у розробленому редакторі конфігурацій елементів інтерфейсу користувача, необхідно оновити відповідне значення у консолі сервісу віддаленої конфігурації Firebase Remote Config. Для цього достатньо вставити значення із буферу обміну, яке ми отримали в п. 4.1.

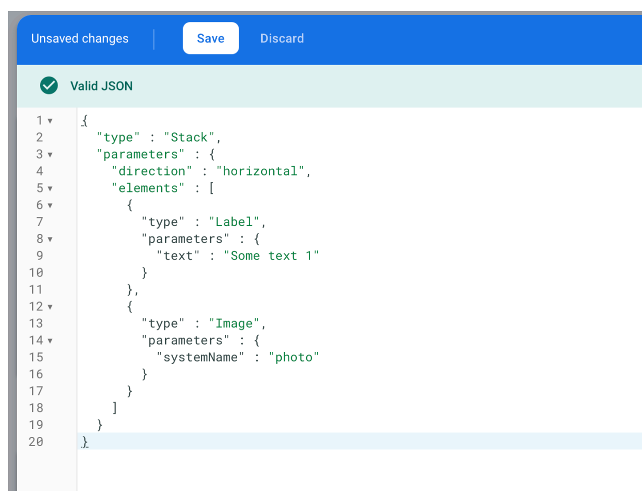


Рисунок 4.8 – додавання створеної конфігурації інтерфейсу користувача у Firebase Remote Config.

Після публікації змін, у користувача автоматично оновлюється інтерфейс додатку, як можна побачити на рис. 4.9.

Оновлені дані конфігурації елементів інтерфейсу користувача отримуються із сервісу в залежності від параметрів SDK Firebase, встановлених користувачем системи, тобто розробником клієнтського додатку. За замовчуванням, конфігурація буде отримана під час кожного нового запуску додакту користувачем, проте, можна налаштувати запити на отримання нових конфігурацій, які більше підходять вимогам, поставленним розробникові (на приклад, під час виклику оновлення користувачем додатку, чи по проходженню

певного періоду часу). Для оптимізації рішення, також, можна встановити кешування конфігурації інтерфейсу користувача.

Окрім цього, SDK Firebase дозволяє завантажувати віддалені конфігурації із локальних джерел (за звичай, .plist файли). Дані джерела будуть використані при запуску додатку до завантаження актуальної віддаленої конфігурації, або коли такі дані не є доступними, як то за відсутності зв’язку із сервісом чи при відключеному зв’язку з інтернетом на пристрої запуску додатку.

Рисунок 4.9 – оновлений інтерфейс клієнтського додатку

Елементи інтерфейсу користувача, отримані із віддалених конфігурації можуть бути використані у різних частнах програми. Розробник, який використовує спроектовану систему сам вирішує методи застосування бібліотеки і не є обмежений засобами її реалізації.

4.3 Розширення системи

Користувач системи, тобто розробник клієнтського додатку, який використовує систему в середовищі iOS може імплементувати власні елементи інтерфейсу користувача, які виконують функції, необхідні умовами задачі розробки. Далі буде розглянуто приклад імплементації такого елементу на прикладі поля вводу повідомлення, із подальшим виведенням написаного користувачем повідомлення на екран пристрою користувача.

Нехай, даний елемент буде мати назву (тип) «AlertableTextInput», і буде приймати такі параметри, як значення плейсхолдеру, розмір шрифта та його колір.

На рисунках 4.10-4.11 наведено реалізацію імплементації даного елементу із розрахунком на розроблену бібліотеку *DecodableUI*.

```
struct AlertableTextInput: View {

    var placeholder: String
    var fontSize: CGFloat
    var fontColor: Color

    var body: some View {
        ContainedView(placeholder: placeholder)
            .font(.system(size: fontSize))
            .foregroundColor(fontColor)
            .textFieldStyle(.roundedBorder)
            .padding()
    }

}
```

Рисунок 4.10 – імплементація прикладового елементу інтерфейсу користувача

Як можна помітити, даний елемент інтерфейсу нічим не відрізняється від вигляду імплементації стандартного елементу із використанням фреймворку SwiftUI, і здатний існувати за межами розробленої системи. Для взаємодії цієї

структури із розробленою системою, необхідно її розшити, надав їй властивостей *DecodableView* із розробленої бібліотеки *DecodableUI*.

```swift
extension AlertableTextInput: DecodableView {

    var anyView: AnyView {
        AnyView(body)
    }

    init?(from decoder: Decoder?, viewResolver: DecodableViewResolver) {
        let container = try? decoder?.container(keyedBy: CodingKeys.self)

        guard
            let placeholder = try? container?.decode(String.self, forKey: .placeholder),
            let fontSize = try? container?.decode(CGFloat.self, forKey: .fontSize),
            let fontColor = try? container?.decodeColorFromHexString(forKey: .fontColor)
        else {
            return nil
        }

        self.placeholder = placeholder
        self.fontSize = fontSize
        self.fontColor = fontColor
    }

    enum CodingKeys: String, CodingKey {
        case placeholder
        case fontSize
        case fontColor
    }

}
```

Рисунок 4.11 – конформація прикладового елементу конфігурації до протоколу *DecodableView*

Далі, необхідно додати створену структуру до реєстру елементів інтерфейсу користувача у класі *DecodableUIProvider* (див. п. 3.4). Після виконання цих дій, можна перейти до розробленого редактору конфігурацій для налаштування параметрів відповідно до поставленої задачі. Розроблена структура, на жаль, повинна потрапити до

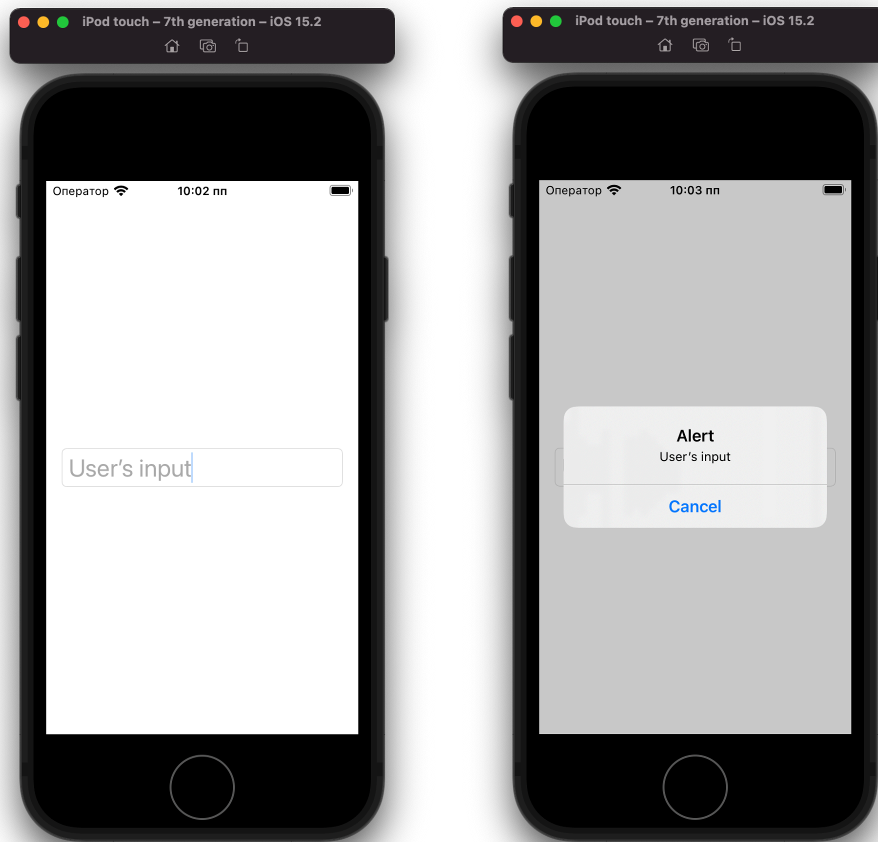
Наступним кроком буде додавання конфігурації елементу у сервіс віддаленого налаштування інтерфейсу. На рисунку 4.12 можна побачити вигляд рядку JSON формату із налаштуваннями заданого елементу інтерфейсу користувача.

```json
{
  "type": "AlertableTextInput",
  "parameters": {
    "placeholder": "Enter text here:",
    "fontSize": 25,
    "fontColor": "#ABABAB"
  }
}
```

Рисунок 4.12 – конфігурація прикладового елементу інтерфейсу користувача у форматі JSON

Після публікації даного запису, на клієнтських додатках з’явиться новий інтерфейс користувача, як зображено на рисунках 4.12-4.13.

Після реалізації прикладового елементу інтерфейсу, використовуючи розроблену систему, можна зробити висновки щодо легкості внесення системи до вже існуючого функціоналу певного клієнтського додатку. Таким чином, розробникам достатньо інтегрувати сервіс віддаленої конфігурації у власний додаток, налаштувати *DecodableViewsResolver* і розширити обрані їм елементи інтерфейсу до протоколу *DecodableView*.



Рисунки 4.12-4.13 – результат реалізації прикладового елементу інтерфейсу користувача

4.4 Рекомендації щодо розвитку та вдосконалення системи

У подальшому для покращення і поглиблення функціональності можна додати декілька напрямків змін. По-перше, покращити роботу редактору налаштувань для підтримки сторонніх конфігураційних елементів інтерфейсу користувача *DecodableView* із зручним варіантом їх імпортування. Такий підхід можна вирішити застосувавши рантайм Objective-C для імпорту та компіляції додатку на етапі його життевого циклу. Інший варіант вирішення проблеми – розбиття редактору на декілька різних додатків, в одному із яких можна імпортувати програмний код, і який буде запускати власне редактор, із вже існуючою реалізацієї обраних елементів інтерфейсу. По-друге, налаштувати сервіс віддаленої конфігурації на надання різних версій конфігурації інтерфейсу користувача в залежності від версії додатку користувача. Дану проблему цілком можна вирішити вбудованим функціоналом сервісу Firebase Remote Config.

ВИСНОВОК ДО РОЗДІЛУ 4

У четвертому розділі представлена система програмних компонентів і підходів взаємодії із нею, розроблені протягом всієї дипломної роботи. Надано загальні вказівки щодо використання системи і її програмних компонентів, а також було розглянуто варіанти розширення системи користувачами для вирішення заданих проблем.

Було досліджено програмний інтерфейс компонентів системи і способи взаємодії із ними.

Далі було розглянуто типові шляхи взаємодії із системою задля вирішення задач проекту.

Після цього було наведено приклад розширення системи її користувачами задля вирішення специфічних проблем, а також, результати такої взаємодії із системою.

В кінці цього розділу було запропоновано можливі ідеї щодо розширення та вдосконалення функціональних можливостей системи, та шляхи їх реалізації, які можуть бути виконані в майбутньому.

ВИСНОВКИ

Під час виконання дипломної роботи було розроблено комплексну систему для реалізації віддаленого налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS. Дана система дозволяє користувачам, тобто розробникам клієнтських додатків для мобільної операційної системи iOS шляхом використання основних методів розробки, змінювати інтерфейс додатку віддалено, без необхідності публікації нової версії застосунку.

У першому розділі було проаналізовано предметну область системи, ключові терміни та область застосунку системи. Були розглянуті і пояснені основні моменти, необхідні для розуміння області застосування системи, а також, проблеми і необхідність рішення, яке може їх розв’язати.

У другому розділі було детально описано вимоги до розроблюваної системи, а також, розглянуто комплекс технологій, підходів та інструментів, які необхідні для реалізації системи відповідно до поставлених вимог. Було проаналізовані дані технології та обґрунтовано вибір засобів реалізації даної системи.

У третьому розділі було описано реалізацію поставленої задачі використовуючи засоби реалізації та стек технологій, наведених у другому розділі. Було наведено ключові об’єкти розробки системи та описано головні моменти їх реалізації. Також, було розглянуто шляхи взаємодії компонентів системи для вирішення поставленої задачі. Окрім цього, було описано процес налаштування та інтеграції таких компонентів системи як сервіс віддаленої конфігурації та розроблену бібліотеку віддалено налаштовуємих елементів інтерфейсу, а також, шлях інтеграції розробленої бібліотеки до редактору налаштувань елементів інтерфейсу користувача.

У четвертому розділі було розглянуто приклади використання системи вцілому, а також, її програмних компонентів окремо. Також, було описано шляхи реалізації розширення системи її користувачами, і наведено приклад такої взаємодії із систему для вирішення прикладової задачі. Окрім цього, в даному розділі було проаналізовано недоліки системи і шляхи до її покращення, а також, наведено можливі виріанти реалізації цих типів покращення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. INC – Why You Need to Follow the Steve Jobs Method and 'Work Backwards' [Електронний ресурс] – Режим доступу до ресурсу: https://www.inc.com/dave-bailey/why-you-need-to-follow-the-steve-jobs-method-and-w.html
2. CNBC – Inside Apple's team that greenlights iPhone apps for the App Store [Електронний ресурс] – Режим доступу до ресурсу: https://www.cnbc.com/2019/06/21/how-apples-app-review-process-for-the-app-store-works.html
3. Firebase Remote Config Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/remote-config
4. The Verge – iOS: A visual history Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad
5. Meta Open Source [Електронний ресурс] – Режим доступу до ресурсу: https://opensource.fb.com/projects
6. AppleInsider – Swift [Електронний ресурс] – Режим доступу до ресурсу: https://appleinsider.com/inside/swift
7. Fireart Studio – React Native vs Swift [Електронний ресурс] – Режим доступу до ресурсу: https://fireart.studio/blog/react-native-vs-swift-which-to-choose-for-an-ios-app/
8. Timac – Apple's use of Swift and SwiftUI in iOS 15 [Електронний ресурс] – Режим доступу до ресурсу: https://blog.timac.org/2021/1219-state-of-swift-and-swiftui-ios15/
9. Firebase Pricing Plans [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/pricing

10. Stackshare – Firebase service [Електронний ресурс] – Режим доступу до ресурсу: https://stackshare.io/firebase
11. Fluid – Mobile Designs 101 [Електронний ресурс] – Режим доступу до ресурсу: https://blog.fluidui.com/designing-for-mobile-101-pixels-points-and-resolutions/
12. Apple Developer – SF Symbols 4 [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/sf-symbols/
13. Firebase Console Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/remote-config/get-started?platform=ios
14. Firebase SDK Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/firebase/firebase-ios-sdk

ДОДАТОК 1

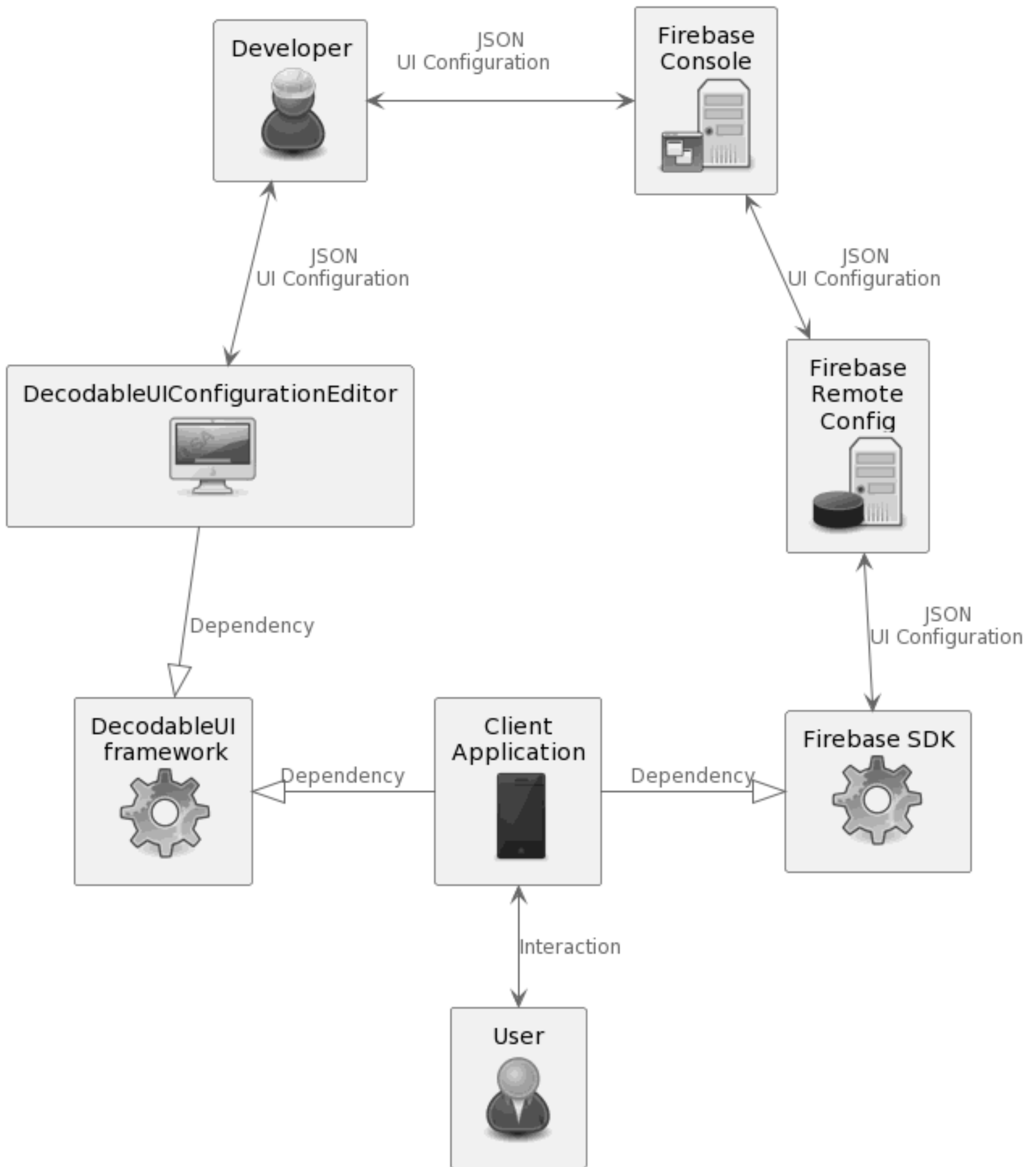
Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1			
		№ докум.	Підпис	Дата				
Розробив		Павловський В.Є.			**Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS**	Літ.	Аркуш	Аркушів
Перевірив							1	1
					Структурна схема системи	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84		
Н. Контр.		Сімоненко В. П.						
Затвердив								

ДОДАТОК 2

Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р

DecodableUI

«protocol» DecodableViewResolver
+ resolve(from:)

«protocol» DecodableView
○ anyView : AnyView
+ init(from:viewResolver:)

«struct» DecodableViewConfiguration
△ type : String
△ nestedDecoder : Decoder?
+ init(from:)

«struct» StackView (Modifier: ViewModifier)
□ direction : Direction
□ spacing : CGFloat?
□ alignment : Alignment
□ elements : [AnyView]
□ viewModifier : Modifier?
○ body : some View
+ init(direction:spacing:alignment:viewModifier:elements:)

«struct» LabelView (Modifier: DecodableViewModifier)
□ text : String
□ fontSize : CGFloat?
□ lineLimit : Int?
□ fontColor : Color?
□ viewModifier : Modifier?
□ font : Font?
○ body : some View
+ init(text:fontSize:lineLimit:fontColor:viewModifier:)

«struct» ImageView (Modifier: ViewModifier)
□ url : URL?
□ systemName : String?
□ viewModifier : Modifier?
○ body : some View
+ init(url:systemName:viewModifier:)

DecodableViewsService
□ viewsRegistry
+ init(viewTypes:)
+ register(_:)

«extension» ImageView
○ anyView : AnyView
+ init(from:viewResolver:)

«extension» LabelView
○ anyView : AnyView
+ init(from:viewResolver:)

«extension» StackView
○ anyView : AnyView
+ init(from:viewResolver:)

«extension» DecodableViewConfiguration

«extension» StackView
□ stack : some View
□ content : some View

SwiftUI

«protocol» ViewModifier
+ body(content:)

«protocol» View
○ body: some View

«extension» DecodableViewsService
+ resolve(from:)

«extension» ImageView
image : some View
systemImage : some View
errorImage : some View
loadingView : some View

«protocol» DecodableViewModifier
+ init(from:)

«struct» DefaultViewModifier
□ background : Color
□ cornerRadius : CGFloat
□ padding : CGFloat?
□ width : CGFloat?
□ height : CGFloat?
+ body(content:)
▲ init(background:padding:cornerRadius:width:height:)

«extension» DefaultViewModifier
+ init(from:)

confirms to · ext · inherits

JSONEditor

DecodableUIProvider
□ viewsService : DecodableViewsService
△ view : AnyView?
△ jsonSubject
△ configuration : DecodableViewConfiguration?
□ cancellables
□ configurationPublisher : AnyPublisher<DecodableViewConfiguration?, Never>
▲ init(viewTypes:)

ObservableObject

JSONModel
△ selectedId : UUID?
△ rootObject
△ cancellables
△ jsonPublisher : AnyPublisher<String, Never>
▲ init()
▲ isRootObject(_:)
▲ create(after:)
▲ delete(with:)
▲ deleteSelected()
■ setSelected(id:)

«struct» JSONHierarchy
△ jsonModel : JSONModel
△ body : some View
■ link(to:)
■ toggleSidebar()

«struct» RecursiveView (Element, RowContent)
△ elements : [Element]
△ children : WritableKeyPath<Element, [Element]>
△ body : some View
▲ rowContent : (Binding<Element>) -> RowContent
■ row(for:)

«struct» JSONRowKeyField
□ key : String?
□ state : String
△ body : some View
▲ init(key:)

«struct» HierarchyList (Element, HeaderContent, RowContent)
□ rootElement : Element
□ isExpanded
△ body : some View
■ rowContent : (Binding<Element>) -> RowContent
■ headerContent : () -> HeaderContent
▲ init(rootElement:children:rowContent:header:)

«struct» JSONRowEditor
□ jsonModel : JSONModel
△ row : JSONRow
□ columns
△ body : some View
□ valueEditor : some View

View

«struct» JSONEditor
□ jsonModel
△ jsonSubject : PassthroughSubject<String, Never>
△ body : some View

«struct» JSONHierarchyRow
△ jsonModel : JSONModel
△ row : JSONRow
△ body : some View

«struct» ContentView
△ uiProvider
△ body : some View

«extension» JSONRow
jsonString : String

«struct» JSONRow
△ id
△ key : String?
△ value : JSONValue
▲ init(key:value:)

«extension» JSONRow
removeRow(with:)
create(after:)
previousWideRow(for:)

«extension» JSONValue
valueDescription : String

«extension» JSONValue
hash(into:)
(_:_:)

«extension» JSONRow
nestedRows : [JSONRow]

«extension» JSONValue
allCases : [JSONValue]

Equatable

CaseIterable

Hashable

Identifiable

«enum» JSONValue
△ null
▲ object(_:)
▲ array(_:)
▲ string(_:)
▲ number(_:)
▲ bool(_:)

«extension» JSONValue
objectValues : [JSONRow]
arrayValues : [JSONRow]
stringValue : String
numberValue : Int
boolValue : Bool

«extension» JSONRow
imageSystemName : String

«extension» JSONValue
jsonString : String

«extension» JSONValue
typeDescription : String

inherits · ext

					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Павловський В.Є.			**Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS**	Літ.	Аркуш	Аркушів
Перевірив							1	2
					Функціональна схема (діаграма класів)	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84		
Н. Контр.		Сімоненко В. П.						
Затвердив								

ДОДАТОК 3

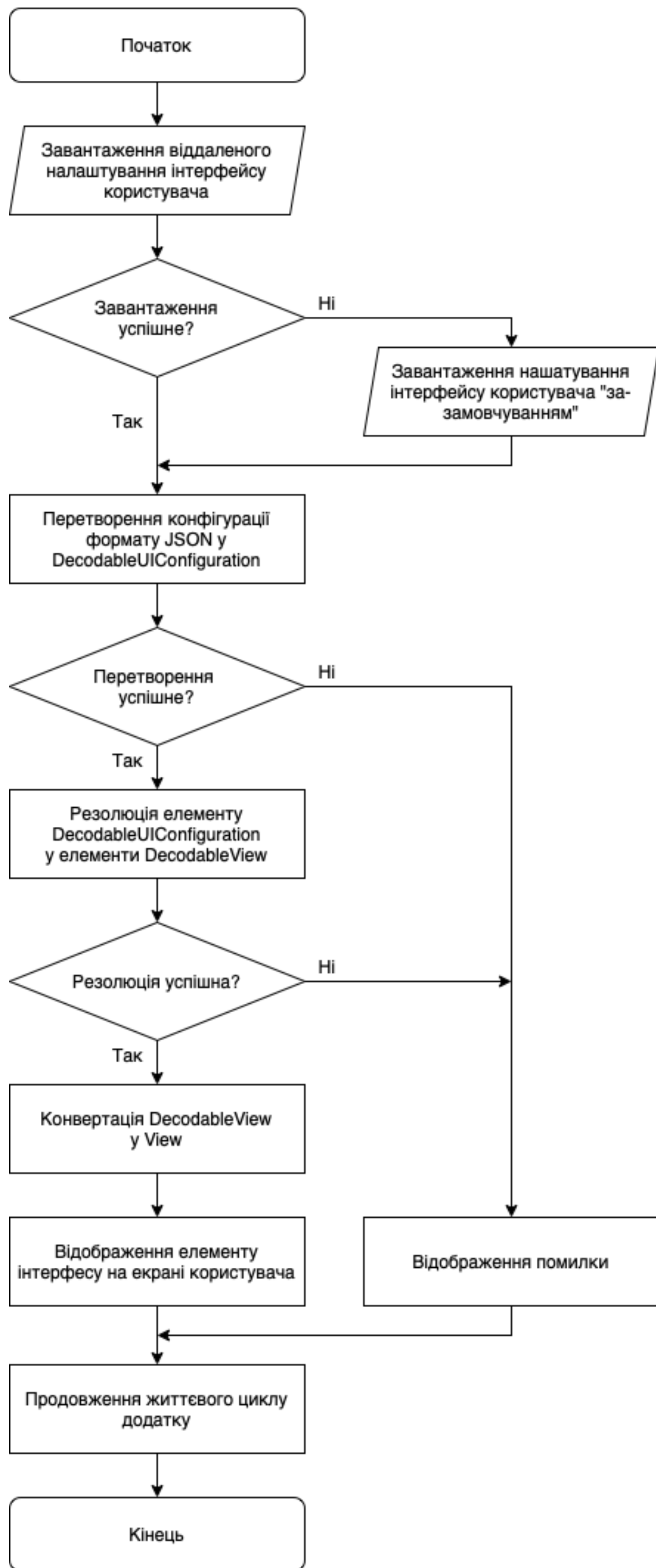
Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 ДЗ			
		№ докум.	Підпис	Дата				
Розробив		Павловський В.Є.			**Віддалене налаштування користувацьких інтерфейсів клієнтських додатків в середовищі iOS**	Літ.	Аркуш	Аркушів
Перевірив							1	1
					Алгоритм дій програм-ного забезпечення	НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84		
Н. Контр.		Сімоненко В. П.						
Затвердив								