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

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
__________ О.В.Коваль
(підпис) (ініціали, прізвище)
“___”_____________2019 р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки
6.050101 “Комп’ютерні науки”

на тему: Система уніфікації методів доступу до баз даних

Виконав: студент 4 курсу, групи ТР-52

Гордієнко Олександр Олександрович __________
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент, к.т.н. Коваль Олександр Васильович __________
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент ст. викладач, к.т.н. Баранюк Олександр Володимирович __________
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____________
(підпис)

Київ – 2019

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки 6.050101 “Комп’ютерні науки”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_______ О.В. Коваль
(підпис)
”___”___________2019 р.

ЗАВДАННЯ
на дипломну роботу студенту

Гордієнку Олександру Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи “Система уніфікації методів доступу до баз даних”

керівник роботи доцент, к.т.н. Коваль Олександр Васильович
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”___”___________ 201__ р. № _________

2. Строк подання студентом роботи ___ ___________ 201__ р.
3. Вихідні дані до роботи персональний комп’ютер під керуванням операційної системи macOS, середовище розробки Xcode, мова програмування Swift.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) аналіз поставленої задачі та можливих варіантів її розв’язку; короткий опис використаних засобів розробки; визначення поняття фреймворку та його різновидів; опис програмної реалізації розробленої системи; тестування розробленого рішення та використані при цьому методології; опис роботи користувача із розробленою системою; висновки за результатами виконаної роботи.
5. Перелік ілюстраційного матеріалу (з точним зазначенням обов’язкових креслень) 1. Фреймворки, бібліотеки та їх види. 2. Опис роботи розробленої системи. 3. Реалізація логіки декодування об’єктів системою. 4. Методологія Test-driven development. 5. Тестування розробленого рішення. 6. Робота користувача із розробленою системою.

Дата видачі завдання ”___”________ 201__ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Вивчення та аналіз задачі		
2.	Розробка архітектури та загальної структури системи		
3.	Підготовка матеріалів		
4.	Програмна реалізація системи		
5.	Захист програмного продукту		
6.	Оформлення пояснювальної записки		
7.	Передзахист		
8.	Захист		

Студент ____________ Гордієнко О.О.
(підпис) (прізвище та ініціали)

Керівник роботи ____________ Коваль О.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Мета роботи — вирішення проблеми роботи вбудованого фреймворку CoreData із протоколом Codable, використовуваних під час десеріалізації даних, отриманих із веб-серверу. Для розв'язку поставленої задачі було використано середовище розробки Xcode та мову програмування Swift. Результатом дипломної роботи є створений динамічний фреймворк CodableMO.framework, який містить у собі протокол DecodableMO, котрий повністю вирішує проблему дублювання об'єктів у базах даних. Розроблене рішення є оптимальним з точки зору використання апаратних ресурсів та належним чином протестоване шляхом написання юніт-тестів.

Записка містить 69 сторінок, 22 рисунки, 3 додатків та 20 посилань.

Ключові слова: APPLE, XCODE, SWIFT, COREDATA, CODABLE, IOS, MACOS, FRAMEWORK, UNIT TESTING, DATABASE.

ABSTRACT

The purpose of the work is to solve the problem of using the built-in CoreData framework with the Codable protocol, used during the deserialization of the data received from the web server. To solve the task, the Xcode development environment and the Swift programming language were used. The result of the diploma work is the dynamic framework CodableMO.framework, that includes the protocol DecodableMO, which completely solves the problem of duplication of objects in the databases. The developed solution is optimal in terms of the use of hardware resources and properly tested by written unit tests.

The note contains 69 pages, 22 pictures, 3 applications and 20 links.

KEYWORDS: APPLE, XCODE, SWIFT, COREDATA, CODABLE, IOS, MACOS, FRAMEWORK, UNIT TESTING, DATABASE.

ЗМІСТ

Перелік умовних позначень, скорочень і термінів 6

ВСТУП 8

1 ПОСТАНОВКА ЗАДАЧІ ВИРІШЕННЯ ПРОБЛЕМИ РОБОТИ ФРЕЙМВОРКУ COREDATA ІЗ ПРОТОКОЛОМ CODABLE 11

2 ЗАСОБИ РОЗРОБКИ 13

2.1 Середовище розробки Xcode 13

2.2 Мова програмування Swift 14

2.3 Фреймворк CoreData 14

2.4 Протокол Codable 16

Висновки до розділу 2 16

3 АНАЛІЗ ПОСТАВЛЕНОЇ ЗАДАЧІ 17

3.1 Причини та наслідки вирішуваної проблеми 17

3.2 Можливі варіанти вирішення проблеми 18

Висновки до розділу 3 19

4 ФРЕЙМВОРКИ, БІБЛІОТЕКИ ТА ЇХ ВИДИ 20

4.1 Поняття фреймворку 20

4.2 Статичні бібліотеки 22

4.3 Динамічні бібліотеки 23

Висновки до розділу 4 25

5 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ 26

5.1 Алгоритм роботи фреймворку 26

5.2 Деталі роботи системи 29

5.3 Гнучкість та незалежність системи 34

Висновки до розділу 5 35

6 ТЕСТУВАННЯ РОЗРОБЛЮВАНОГО РІШЕННЯ 36
6.1 Методологія Test-driven development 36
6.2 Тестування поведінки системи 40
6.3 Тестування вирішення проблеми дублювання 42
6.4 Тестування контекстної незалежності 43
6.5 Тестування впливу на продуктивність 44
Висновки до розділу 6 46
7 РОБОТА КОРИСТУВАЧА ІЗ РОЗРОБЛЕНОЮ СИСТЕМОЮ 47
7.1 Системні вимоги 47
7.2 Робота розробника зі створеним фреймворком 47
7.3 Приклади роботи із фреймворком 50
Висновки до розділу 7 52
ВИСНОВКИ 53
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 54
Додаток А 56
Додаток Б 58
Додаток В 61

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

IDE — Integrated Development Environment, комплексне програмне рішення для розробки програмного забезпечення.

API — Application Programming Interface, набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

SDK — Software Development Kit, набір із засобів розробки, утиліт і документації, який дозволяє програмістам створювати прикладні програми за визначеною технологією або для певної платформи (програмної або програмно-апаратної).

Mach-O — Mach object, файловий формат для запускаючого та об'єктного коду, динамічних бібліотек та дампів пам'яті, котрий використовують операційні системи Apple Inc.

SQL — Structured query language, декларативна мова програмування для взаємодії користувача з базами даних, що застосовується для формування запитів, оновлення і керування реляційними БД, створення схеми бази даних та її модифікації, системи контролю за доступом до бази даних.

NeXT — NeXT Software, Inc, американська комп'ютерна компанія, яка розробляла і виробляла графічні робочі станції, призначені для ВНЗ та бізнесу.

OpenStep — об'єктно-орієнтований інтерфейс програмування додатків (API) для об'єктно-орієнтованих операційних систем, які використовують будь-яку сучасну операційну систему у якості ядра.

EOF — Enterprise Objects Framework, продукт для об'єктно-реляційного відображення для платформ розробки NeXTSTEP та OpenStep.

WWDC — Worldwide Developers Conference, щорічна конференція розробників для платформи Macintosh.

JSON — JavaScript Object Notation, текстовий формат обміну даними між комп'ютерами, базується на тексті, може бути прочитаним людиною.

Nib-файл — опис фрагменту користувацького інтерфейсу у скомпільованому форматі файлу із розширенням “.nib”.

LLDB — зневаджувач, заснований на технологіях проекту LLVM. LLDB являє собою інфраструктуру для організації зневадження застосунків і складається з набору модулів і бібліотек.

TDD — Test-driven development, технологія розробки програмного забезпечення, яка використовує короткі ітерації розробки, що починаються з попереднього написання тестів, які визначають необхідні покращення або нові функції.

ВСТУП

Зараз така епоха, коли смартфони відіграють неоцінимо важливу роль у житті людей. Вони замінили майже все: раніше для того, щоб дізнатися час, потрібно було або мати годинник, або постійно запитувати у когось, котра година; новини та прогноз погоди ми звикли дізнаватися з відповідних передач по телебаченню; для того, щоб дістатися у якусь точку країни, ми, зазвичай, використовували паперові карти. Це лише найменша частина з того, що зараз можна зробити у пару дотиків, використовуючи найрізноманітніші мобільні додатки.

Так вже сталося, що в останнє десятиліття IT-індустрія стала дуже важливою та впливовою галуззю у світі, і, особливо, в Україні, і тому, у часи активного розвитку клієнт-серверних продуктів, людство має неабиякі можливості, маючи всього лише смартфон із підключенням до інтернету та встановленими на ньому необхідними програмними додатками.

Сьогодні безкоштовними відео дзвінками на другу частину земної кулі вже нікого не здивуєш, технології розвиваються настільки швидко, що важко навіть передбачити, чим найталановитіші дизайнери та розробники світу можуть здивувати вже завтра.

Тут дуже важливо зауважити, що у процесі розробки програмних додатків, на етапі проектування системи, інженери також оцінюють можливі обмеження, які перед ними можуть з’явитися. Ця оцінка має великий вплив на те, який набір інструментів буде використовуватися під час реалізації поставленої задачі, адже, як відомо, кожен інструмент має свої, як переваги, так і недоліки.

Одним із основних таких обмежень, як не дивно, є наявність інтернет-з’єднання. Адже, дійсно, ситуація із неякісним з’єднанням, або його відсутністю, є досить неприємною, і якщо, наприклад, при достатньо слабкому підключенні доводиться значно довше чекати на відповідь від веб-серверу, то при його відсутності, у деяких випадках, робота із додатком стає попросту неможливою.

Будь-який досвідчений інженер відразу ж запропонує вирішення даної проблеми – достатньо просто зберігати відповіді веб-серверу у локальній базі даних, а у випадку відсутності з’єднання з ним – повторити надісланий запит, як тільки з’єднання з’явиться, паралельно інформуючи користувача про тимчасові обмеженості. З цим майже неможливо не погодитися, тим паче, що крім цього ми отримаємо вирішення ще однієї достатньо розповсюдженої проблеми — очікування інформації, яка раніше вже отримувалася (ця проблема є особливо актуальною, якщо розмір очікуваних даних достатньо великий). Після такої пропозиції, очікувано, починається вибір відповідного інструментарію для кожної із задіяних платформ.

Якщо, наприклад, говорити про платформу iOS, то вибір бібліотек, які дозволяють створювати та керувати локальними базами даних, достатньо великий: Realm, SQLite, CoreData. Явним фаворитом, особливо для мене, серед цих трьох бібліотек є CoreData. Вона має достатньо багато переваг, основними серед них є: гарна документованість, легкість у використанні, легкість у міграції (найпростіша міграція навіть не потребує втручання розробника). Головною ж перевагою є те, що ця бібліотека є дуже швидкою та оптимізованою, оскільки вона була написана інженерами тієї ж компанії, котра, власне, і розробляла саму платформу iOS.

У зауваженої бібліотеки є лише один, але дуже серйозний недолік. Під час десеріалізації інформації, отриманої від веб-серверу, за допомогою використання протоколу Decodable, у випадку, якщо такий об’єкт вже існує у локальній базі даних, буде створено його дублікат. Одне з можливих пояснень — для таблиць баз даних, створених за допомогою бібліотеки CoreData, неможливо вказати первинні ключі, і, через це, об’єкти, котрі додаються до бази, не перевіряються на дублювання.

Вказана проблема змушує розробників вигадувати усілякі шляхи для її уникнення (які, як правило, не є оптимальними з точки зору продуктивності), або, взагалі, змінити свій вибір на користь інших запропонованих бібліотек.

Вирішення вище зазначеної проблеми допоможе заощадити стороннім розробникам час на пошуки, тестування та налагодження оптимального розв’язку для даної задачі, та може бути перевикористане у будь-яких інших проектах.

Пояснювальна записка до дипломної роботи містить 7 розділів.

У першому розділі описується постановка задачі вирішення проблеми роботи фреймворку CoreData із протоколом Codable.

У другому розділі описуються основні засоби розробки для даної системи.

У третьому розділі проводиться аналіз поставленої задачі: причини проблеми та можливі варіанти їх вирішення.

У четвертому розділі дано детальний опис поняттю “фреймворк”, видів бібліотек, які можуть в ньому міститися, та їх різницю.

У п’ятому розділі дано опис реалізованого програмного продукту та алгоритму його роботи.

У шостому розділі вказано способи тестування продукту та основної методології, яка при цьому використовувалася.

У сьомому розділі описано роботу користувачів із розробленою системою.

1 ПОСТАНОВКА ЗАДАЧІ ВИРІШЕННЯ ПРОБЛЕМИ РОБОТИ ФРЕЙМВОРКУ COREDATA ІЗ ПРОТОКОЛОМ CODABLE

Використовуючи теоретичний матеріал, наданий керівником дипломної роботи, реалізувати бібліотеку, яка вирішить проблеми використання вбудованої в iOS SDK бібліотеки CoreData у парі з протоколом Codable, використовуваних при десеріалізації інформації, отриманої з веб-серверу.

Розроблена система повинна задовольняти наступні критерії:

— повинна бути у вигляді динамічної бібліотеки;
— мати оптимальну реалізацію із точки зору впливу на продуктивність;
— мати гарну документованість, дотриману стилю документації від Apple;
— відсутність різниці у поведінці у порівнянні із вбудованим функціоналом – протоколом Codable;
— успішно проходити тести на десеріалізацію даних різних структур та тести на уникнення дублювання;
— бути максимально гнучкою та незалежною, не повинна мати обмежень, з якими б стикнулися сторонні розробники.

Для вирішення поставленої задачі використовувати операційну систему macOS, середовище розробки Xcode та мову програмування Swift. Розроблений продукт не повинен мати графічного інтерфейсу, так як він має бути суто динамічною бібліотекою, котра вирішує логічний конфлікт двох вбудованих в iOS SDK потужних інструментів. Кінцевими користувачами вище описаної системи є сторонні розробники, котрі, в силу поставлених перед ними задач, зіткнулися із зазначеною проблемою.

Метою вирішення поставленої задачі є створення динамічної бібліотеки, котра допоможе стороннім розробникам заощадити цінний час на пошуки власного рішення цієї проблеми, або на вивчення переваг та недоліків запропонованих аналогів.

Крім того, вирішення вказаної задачі шляхом створення саме фреймворку дозволить замінити непотрібне дублювання коду у всіх проектах, які цього потребують, на звичне усім інженерам підключення сторонньої бібліотеки, котре виконується у декілька натисків лівої кнопки миші.

2 ЗАСОБИ РОЗРОБКИ

Дуже важливою частиною перед безпосередньою реалізацією вирішення поставленої задачі є вибір набору інструментів, які будуть використовуватися протягом цього непростого процесу.

Якщо, наприклад, говорити про задачі обчислення та візуалізації, то вибір інструментів є настільки широким, що у процесі підбору можна враховувати, навіть, власні примхи. Поставлена ж задача чітко обмежує нас рамками середовища розробки Xcode та мови програмування Swift, оскільки вказана проблема виникає при взаємодію протоколу Codable із бібліотекою CoreData, що можливо лише при розробці із використанням зазначеного середовища розробки та мови програмування Swift.

2.1 Середовище розробки Xcode

Середовище Xcode – це інтегроване середовище розробки для операційної системи macOS, котре містить набір інструментів для розробки, виробництва компанії Apple, яке надає можливості для розробки програмного забезпечення для платформ iOS, macOS, tvOS та watchOS [1].

Середовище розробки підтримує розробку програмних продуктів з використанням наступних мов: AppleScript, Objective-C, Swift, Ruby, Python та Java. Крім цього, Xcode вміє створювати, так звані, “fat binary files”, котрі містять код для багатьох архітектур, із файловим форматом для запускаючого та об’єктного коду – Mach-O. Вони називаються універсальними двійковими файлами, котрі дозволяють програмному забезпеченню працювати як на платформах PowerPC, так і на Intel (x86), а також можуть включати в себе як 32-бітний, так і 64-бітний розрядний код для обох архітектур [2].

Разом із iOS SDK, Xcode IDE також може бути використаним для компіляції та налагодження програмного забезпечення для платформи iOS, яке працює на процесорах архітектури ARM [3].

2.2 Мова програмування Swift

Мова Swift — це відкрита, універсальна, мультипарадигмова, компільована мова програмування, розроблена інженерами компанії Apple як засіб розробки програмного забезпечення для платформ iOS, macOS, tvOS та watchOS [4].

Ця мова запозичила достатньо багато із Objective-C, проте, визначається вона не вказівниками, а саме типами змінних, котрі опрацьовує компілятор, за таким же принципом працюють працює більшість скриптових мов. В той же час, Swift пропонує розробникам багато з функцій, які до цього були доступними в мовах C++ та Java, як, наприклад, узагальнене програмування або перевантаження операторів [5].

Частина функцій мови виконується набагато швидше у порівнянні з іншими мовами програмування, так, наприклад, сортування комплексних об’єктів виконується в 3,9 разів швидше, аніж у Python та майже в 1,5 рази швидше, ніж в Objective-C [6].

Дуже важливо зауважити, що код, написаний мовою Swift може без проблем працювати разом із кодом, написаним за допомогою мов C та Objective-C у рамках одного і того ж проекту. Саме ця перевага дає можливість без проблем використовувати бібліотеки, написані мовами Objective-C без потреби переписування їх мовою Swift[7].

2.3 Фреймворк CoreData

Фреймворк CoreData — це фреймворк від компанії Apple, вбудований в операційні системи iOS та macOS, який дозволяє розробнику взаємодіяти з базами даних. Був представлений компанією Apple під час анонсу Mac OS X 10.4 Tiger та

iOS з iPhone SDK 3.0. Керування даними може бути організовано за допомогою маніпуляції сутностей та їх взаємозв'язків [8].

Фреймворк побудовано на трьох структурних елементах:

– Managed Object Context – компонент, з котрим іде взаємодія, кожного разу, коли відбувається будь-яка подія (запис, збереження, редагування, видалення);
– Persistent Store Coordinator – виконує ціль зберігання даних;
– Managed Object Model – являє собою модель бази даних;
– Persistent Store – репозиторій, де зберігаються дані.

Зазначений фреймворк описує дані, котрі зберігаються в додатку, код може маніпулювати для збереження і запису даних у додатку. Модель бази даних створюється в Interface Builder. Код для керування базою даних пишеться на одній з мов: Swift або Objective-C. Сам фреймворк організований у набори величезних класів, які і використовуються для керування усіма потужностями бібліотеки [9].

Використання такого фреймворку є необхідним, якщо потрібно зберігати дані при виході з додатку, тобто, у випадках, коли необхідно відновлювати інформацію з попередніх сесій роботи із додатком. У такому випадку неможливо уникнути роботи із локальною базою даних, оскільки при завершенні сесії всі дані буде втрачено. Вирішення цієї проблеми і є головною ціллю даного фреймворку.

Фреймворк CoreData може конвертувати дані в XML, бінарний код, SQLite для зберігання. У випадках, якщо на комп'ютері не встановлено Xcode, можливість прочитати модель бази даних все одно є [10].

Основна частина фреймворку була написана в часи роботи Стіва Джобса в компанії NeXT, Enterprise Objects Framework (EOF) мовою Objective-C.

Продукт EOF був спеціально розроблений на object-relation mapping для SQL database двигунів як Microsoft SQL Server та Oracle [11].

На сьогоднішній день фреймворк CoreData є найголовнішим в компанії та користується величезним попитом серед сторонніх розробників.

2.4 Протокол Codable

Протокол Codable – тип, який може бути перетворений у зовнішнє представлення та, навпаки, може бути отриманим із зовнішнього представлення.

Тип було представлено на Worldwide Developers Conference (WWDC) 2017 разом з Xcode 9.0. Головною його задачею є полегшення процесу серіалізації та десеріалізації даних при обміні інформацією із веб-сервером.

Протокол Codable складається з двох протоколів Encodable та Decodable.

Протокол Encodable використовується для процесу серіалізації даних, має всього лиш один метод, котрий має реалізувати клас, який слідує цьому протокол.

Протокол Decodable ж, у свою чергу, використовується для процесу десеріалізації даних, і також має лише один метод, який повинен бути реалізованим [12].

Висновки до розділу 2

У даному розділі було коротко описано набір основних інструментів, котрі використовувалися під час розробки системи. Кожен із наведених засобів був необхідним та незамінним під час розв’язання поставленої задачі.

3 АНАЛІЗ ПОСТАВЛЕНОЇ ЗАДАЧІ

3.1 Причини та наслідки вирішуваної проблеми

У описаного вище фреймворку CoreData є дуже велика кількість переваг над існуючими аналогами, тому підстави для відмови від його використання мають бути достатньо вагомими та обґрунтованими.

Однією, можливо і єдиною, такою причиною є проблема використання цієї бібліотеки у парі з протоколом Codable під час десеріалізації даних, отриманих під час обміну інформацією з веб-сервером.

Ідентифікувати причину достатньо просто: під час десеріалізації об’єкту за допомогою протоколу Codable у відповідного класу викликається реалізований метод (конструктор) зазначеного протоколу. Викликаючи конструктор, при цьому не перевіряючи наявність такого об’єкту у базі даних, автоматично створюється новий об’єкт, який, у випадку наявності такого ж об’єкту у базі, буде дублікатом.

Проблема тут саме у тому, що конструктор відповідного класу викликається неявно, у будь-якому випадку, без перевірки на можливість дублювання. Саме тому, при отриманні від веб-серверу об’єкту, який вже міститься у локальній БД, ми створюємо черговий його дублікат. В такому випадку, при відображенні переліку об’єктів такого типу, є стовідсоткова ймовірність наявності однакових об’єктів, при чому, їх кількість абсолютно точно відповідає кількості разів, скільки цей об’єкт було отримано від веб-серверу.

Можливим коренем цієї проблеми є відсутність можливості вказати первинний ключ для будь-якої з таблиць бази даних, створеної за допомогою фреймворку CoreData [8].

Така проблема змушує сторонніх розробників шукати усілякі варіанти для вирішення даної проблеми, або ж, взагалі, відмовитися від використання даного фреймворку на користь інших запропонованих альтернатив.

Варто зазначити, що вище описана проблема досі не була вирішеною, тому створена система є, деякою мірою, унікальною та надає можливість стороннім розробникам без проблем використовувати фреймворк CoreData разом із створеним протоколом CodableMO, котрий за своєю поведінкою ідентичний протоколу Codable.

3.2 Можливі варіанти вирішення проблеми

В інтернеті є чимало статей, котрі демонструють, як легко “потоваришувати” фреймворк CoreData із протоколом Codable, проте майже у кожній з них ані слова не сказано про можливі проблеми із дублюванням існуючих об’єктів.

Після вивчення всіх можливих варіантів, можемо дійти висновку, що серед запропонованих за увесь час розв’язків даної задачі є лише один можливий спосіб уникнути дублювання, всі останні запропоновані варіанти, так чи інакше, в кінцевому випадку зводяться, до нього ж.

Суть такого рішення достатньо проста: потрібно створити структуру, котра буде мати точно такі ж поля, як і сутність моделі бази даних, котра буде десеріалізуватися у процесі обміну інформації з веб-сервером. Створена структура і буде використовуватися під час зазначеного процесу, а вже після нього відбудеться запит до бази даних, котрий і визначить, чи існує в ній такий об’єкт (і його варто всього лиш оновити актуальними даними), чи такий об’єкт необхідно створити.

Проблемою усіх таких варіантів є те, що незалежно від сутностей, які будуть задіяні у процесі обміну інформацією, є необхідність створювати структури, які будуть повністю їх дублювати. При цьому, протягом процесу десеріалізації, у будь-якому випадку буде створено об’єкт такої структури-прокладки. Проблемою у цьому випадку є те, що після цього буде створено ще один такий же об’єкт, якому буде переприсвоєно всі декодовані властивості, або ж буде оновлено до цього існуючий об’єкт, знову ж таки, завдяки переприсвоєнню усіх його властивостей.

Крім цього, необхідно написати допоміжній клас, або, навіть класи, які будуть відповідати за вище описану логіку.

Усі запропоновані варіанти відрізняються лише оптимальністю вище описаного алгоритму. Деякі з рішень основані на використанні шаблонних класів, що значно зменшує кількість коду, необхідного для реалізації розв’язку задачі, але, при цьому ж, ніяк не уникають необхідності створення структур-прокладок, що, насправді, не є обов’язковим.

Висновки до розділу 3

У даному розділі наведено основну причину розв’язуваної проблеми та проаналізовано основні пропозиції щодо її вирішення. Проведений аналіз зазначених пропозицій допоміг уникнути основних помилок, які допускаються при розв’язуванні задач такого типу.

4 ФРЕЙМВОРКИ, БІБЛІОТЕКИ ТА ЇХ ВИДИ

4.1 Поняття фреймворку

Фреймворком називають структурований каталог, котрий містить спільну статичну або динамічну бібліотеку разом із відповідними ресурсами, такими як nib-файли, зображення, аудіо файли і файли заголовків (рисунок 4.1).

Рисунок 4.1 – Структура та взаємодія фреймворку із програмним продуктом

Під час розробки програмного продукту, розроблюваний проект посилається на один або кілька фреймворків. Наприклад, проекти додатків iPhone за замовчуванням пов'язуються з фреймворками Foundation, UIKit і Core Graphics. Відповідний програмний код отримує доступ до можливостей фреймворку через інтерфейс прикладного програмування (API), який описується фреймворком через файли заголовків.

У випадку динамічного фреймворку, оскільки бібліотека спільно використовується динамічно, кілька програм можуть одночасно отримувати доступ до коду фреймворку та його ресурсів. Система завантажує код і ресурси фреймворку в пам'ять, за потреби, і розділяє одну копію ресурсу між усіма додатками.

Оскільки фреймворк по своїй структурі нагадує пакунок (англ. bundle), доступ до його вмісту можна отримати за допомогою класу NSBundle або, для процедурного коду, CFBundle із Core Foundation . У сторонніх розробників є можливість створювати

власні фреймворки для усіх платформ компанії Apple: iOS, macOS, watchOS та tvOS. Використовуючи операційну систему macOS є можливість переглянути вміст фреймворку через додаток Finder. Також, під час розробки фреймворку для будь-якої з платформ, у розробника є можливість переглядати файли заголовків фреймворку із середовища розробки Xcode [13].

Фреймворк є ієрархічним каталогом, який інкапсулює спільні ресурси, котрі у ньому містяться. Однак, на відміну від більшості “пакунків”, пакет не відображається додатком Finder як непрозорий файл. Пакет фреймворку — це стандартний каталог, яким користувач може переміщатися. Це полегшує розробникам перегляд його вмісту й перегляду будь-якої документації та заголовкових файлів.

Вони слугують тій самій меті, що і статичні та динамічні спільні бібліотеки, тобто надають бібліотеку підпрограм, які можна викликати із додатку для виконання певного завдання. Наприклад, фреймворки Application Kit і Foundation надають програмні інтерфейси для класів і методів Cocoa API. Фреймворки пропонують такі переваги перед бібліотеками, пов'язаними з статикою, та іншими типами динамічних спільних бібліотек:

– групи фреймворків пов'язані, але розділені за спільними ресурсами. Таке угруповання полегшує встановлення, видалення та пошук цих ресурсів;

– фреймворки можуть включати більш широке різноманіття типів ресурсів, аніж бібліотеки. Наприклад, фреймворк може включати будь-які відповідні файли заголовків та документацію;

– кілька версій фреймворку можуть бути включені в один і той же пакет. Це робить можливим зворотну сумісність зі старими програмними продуктами;

– тільки одна копія ресурсів, доступних тільки для читання, фізично знаходиться у пам'яті в будь-який момент часу, незалежно від того, скільки процесів використовують ці ресурси. Подібні оптимізаційні рішення значно зменшують кількість необхідної системної пам'яті та підвищують продуктивність [14].

4.2 Статичні бібліотеки

Статичною бібліотекою називають сукупність об'єктних файлів. У свою чергу, об'єктний файл — це лише назва для файлу, котрий виходить із компілятора і містить у собі машинний код.

Статичні бібліотеки мають розширення ".a" і створюються за допомогою інструменту архіватора. Якщо це звучить дуже схожим на ZIP-архів, то саме це і є: статичну бібліотеку можна вважати архівом декількох об'єктних файлів.

Файли об'єктів мають формат Mach-O, який є спеціальним форматом файлів для операційних систем iOS і macOS. Це, у загальному випадку, двійковий потік із наступними фрагментами:

– заголовок: визначає цільову архітектуру файлу. Оскільки один Mach-O містить код і дані для однієї архітектури, код, призначений для архітектури x86-64, не буде працювати на архітектурі arm64.

– команди завантаження: вказують на логічну структуру файлу у вигляді розташування в таблиці символів.

– необроблені дані сегменту: містять необроблений код і дані.

На рисунку 4.2 зображено спосіб, у який спосіб статична бібліотека завантажується в адресний простір для подальшого використання. Таким чином, вони стають частиною виконуваного файлу і статично пов'язані з клієнтськими додатками. Такий підхід змушує розроблену бібліотеку постійно завантажуватися у адресний простір кожного разу із завантаженням туди програмного додатку, при цьому, якщо одночасно буде завантажено декілька додатків, така бібліотека у адресному просторі буде присутня також у кількох екземплярах, хоча усі з них будуть мати ідентичну структуру та реалізацію. При цьому, варто зауважити, що, не дивлячись на багаторазове завантаження до пам’яті, в залежності від функціоналу бібліотеки, можливий сценарій, у котрому жоден із використовуючих додатків може і не скористатися послугами такої бібліотеки [15].

У випадках, коли бібліотека завантажується у адресний простір разом із додатком, який її використовує, час запуску такого додатку значно збільшується, тому велика кількість статичних бібліотек, використовуваних із одного додатку, дуже сильно впливаю на час його завантаження та кількість використовуваної ним пам'яті пристрою.

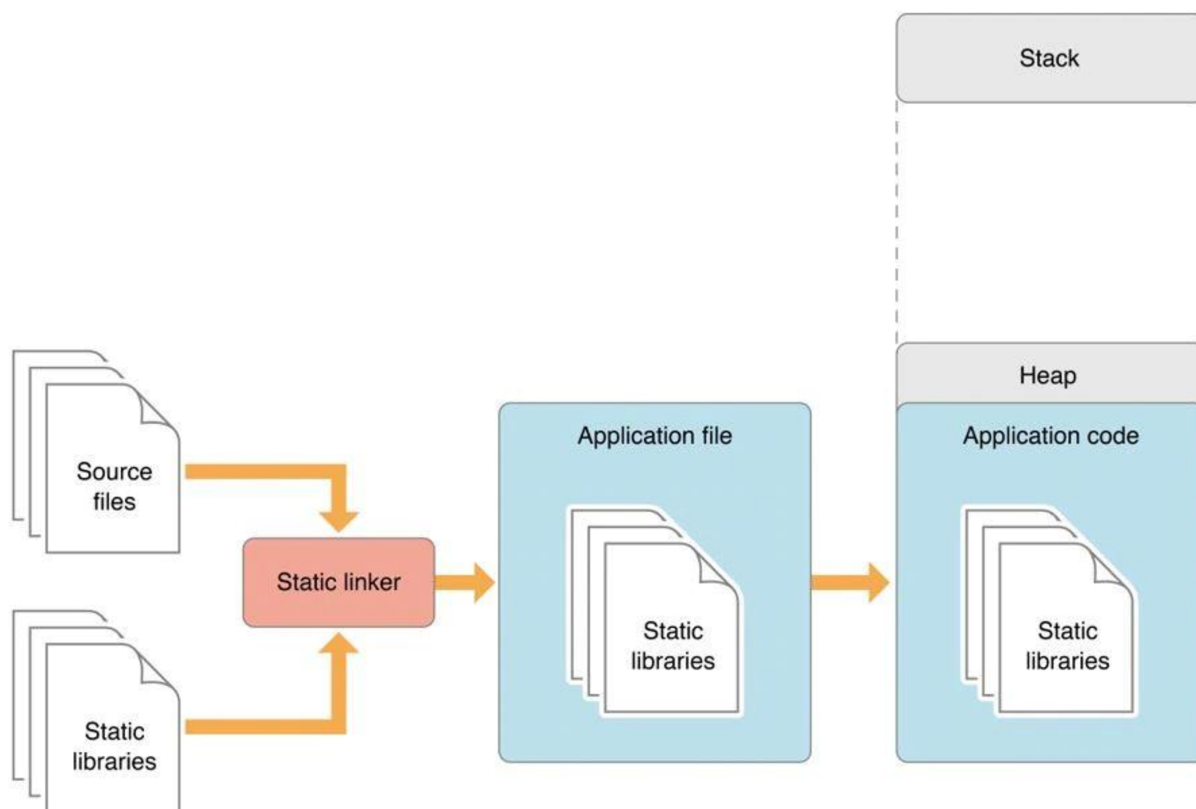

Рисунок 4.2 – Схема процесу завантаження статичної бібліотеки до адресного простору

Описаний вище спосіб не є оптимальним з точки зору використання апаратних ресурсів, особливо у випадках, коли розроблена бібліотека використовується одночасно у декількох додатках.

4.3 Динамічні бібліотеки

Основною динамічних бібліотек є вирішення проблеми оптимальності використання апаратних ресурсів, властивих статичним бібліотекам.

Динамічні бібліотеки, на відміну від статичних, замість того, щоб копіюватися в єдиний монолітний виконуваний файл, завантажуються в пам'ять, коли вони дійсно потрібні (рисунок 4.3).

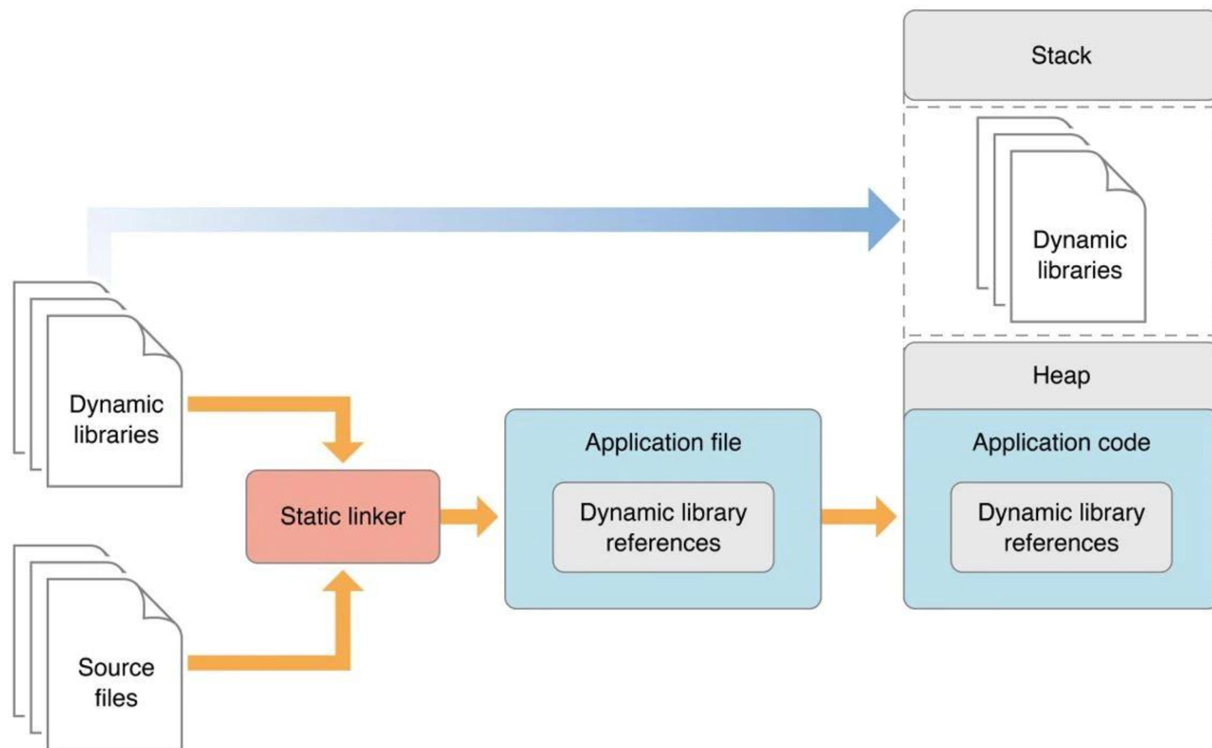

Рисунок 4.3 – Схема процесу завантаження динамічної бібліотеки до адресного простору

Динамічні бібліотеки, як правило, поділяються між додатками, тому системі достатньо зберігати у адресному просторі лише одну копію такої бібліотеки і надавати доступ до неї із різних процесів. Як результат, виклик коду та використання даних із динамічних бібліотек відбувається дещо повільніше, аніж зі статичних.

Тут варто зауважити, що час на виклик коду та використання даних із такої бібліотеки залежить від того, скільки процесів одночасно намагається скористатися її функціоналом. Цей недолік повністю нівелюється тим фактом, що динамічні бібліотеки завантажують у адресний простір лише за їх необхідності, що, у випадку масивних бібліотек, значно зменшує час запуску програмного додатку.

Використовуючи динамічні бібліотеки, програми можуть отримувати вигоду від вдосконалення бібліотек, які вони використовують автоматично, оскільки їхні

посилання на бібліотеки є динамічними, а не статичними. Тобто, функціональність клієнтських додатків може бути покращена і розширена, не вимагаючи від розробників додатків перекомпілювати їх продукти. Усі додатки, написані для macOS, користуються цією функцією, оскільки всі системні бібліотеки в macOS є динамічними. Таким чином програми, які використовують технології Carbon або Cocoa, отримують переваги від покращення OS X.

Єдина річ, яку розробники повинні пам'ятати при розробці динамічних бібліотек — це сумісність із клієнтськими програмами, коли бібліотека оновлюється. Оскільки бібліотеку можна оновити без відома розробника клієнтського додатку, продукт повинен мати можливість використовувати нову версію бібліотеки без змін у її коді. З цією метою API бібліотеки не повинен змінюватися. Тим не менш, є певні моменти, коли покращення вимагають змін API. У цьому випадку попередня версія бібліотеки повинна залишатися на пристрої користувача, щоб клієнтська програма працювала належним чином [15].

Висновки до розділу 4

У даному розділі наводиться детальний опис поняття “фреймворк”, його структури та призначення. Крім того, у розділі наведено основну перевагу динамічних бібліотек над статичними, адже однією з вимог до розроблюваної системи є її реалізація саме у вигляді динамічного фреймворку.

5 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

5.1 Алгоритм роботи фреймворку

Основною частиною розробленої бібліотеки є протокол DecodableMO, який і відіграє головну роль у вирішенні поставленої задачі. Сам протокол складається із конструктора, який буде використовуватися для створення нового об'єкту в базі даних, та двох методів: методу оновлення існуючого в базі об'єкту та методу, котрий повертає об'єкт класу NSPredicate, який буде використовуватися для визначення присутності декодованого об'єкту у БД. Блок-схема поведінки система зображена на рисунку 5.1.

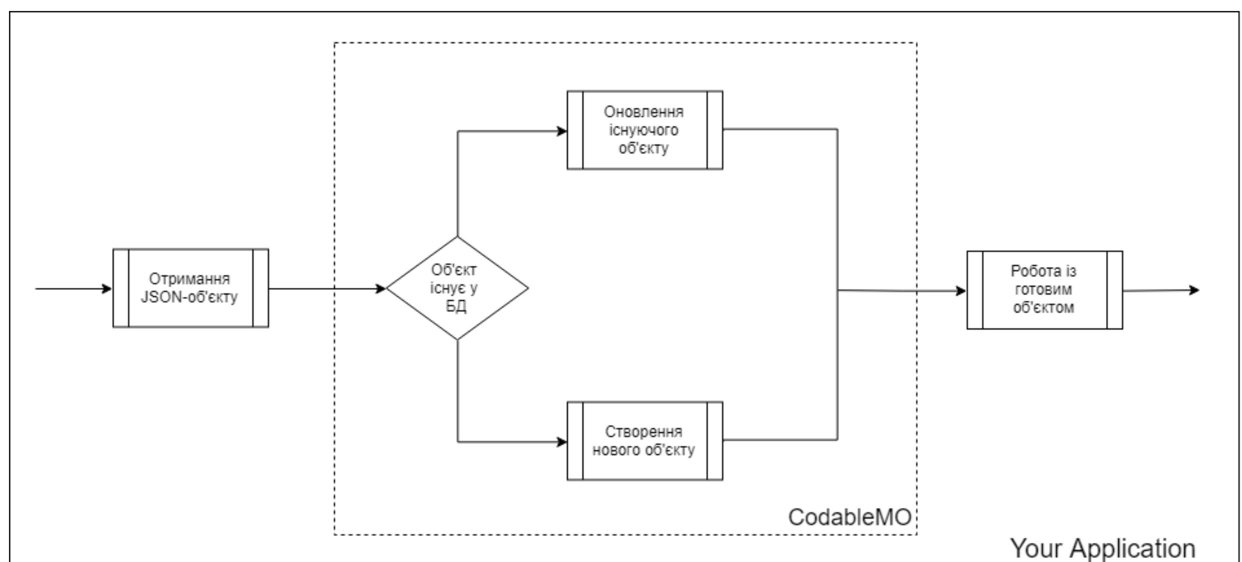

Рисунок 5.1 – Блок-схема алгоритму роботи розробленої системи

Якщо не вдаватися у деталі реалізації, то алгоритм роботи розробленої бібліотеки достатньо простий та очевидний: відразу після конвертації отриманої інформації із масиву байтів до контейнеру, з якого відповідні дані можна діставати по ключам, відбувається запит до бази даних із використанням предикату, який було отримано за допомогою методу, який імплементують усі, хто реалізує протокол DecodableMO. Якщо у БД нема об'єкту, який задовольняє умови запиту, викликається

конструктор, знову ж таки, протоколу DecodableMO, за допомогою якого буде створено новий об'єкт, котрий згодом потрапить до БД. Якщо ж, навпаки, у базі існує об'єкт, який задовольняє надісланий запит (а, згідно алгоритму, таких об'єктів не може бути більше одного), він оновлюється за допомогою останнього методу протоколу DecodableMO.

З точки зору сторонніх розробників, робота із зазначеним протоколом майже не відрізняється від роботи зі звичайним протоколом Codable, а поведінка цих протоколів цілком ідентична.

Реалізовано такий алгоритм за допомогою створення шаблонних обгорток над класами, які реалізують протокол DecodableMO. Це дало змогу отримати доступ до вище згаданого декодованого контейнеру з ключами ще на етапі до виклику конструктора відповідних класів. Приклад одного з таких класів-обгорток можна побачити на рисунку 5.2.

```
class MOModelWrapper<T>: Decodable where T: DecodableMO {
    var model: T

    required init(from decoder: Decoder) throws {
        let context = try decoder.getContext()
        var encodedObject = try decoder.container(keyedBy: T.MOCodingKeys.self)
        self.model = try T.getObject(with: &encodedObject, in: context)
    }

}
```

Рисунок 5.2 – Приклад класу-обгортки для протоколу DecodableMO

Відрізняються ж такі обгортки лише тим, у якій формі зберігаються об'єкти відповідних класів: одиничний об'єкт, масив об'єктів одиничної вкладеності та у вигляді масиву невизначеної структури та вкладеності.

Якщо заглибитись у деталі реалізації, то можна побачити використання малопопулярного API, тісно пов'язаного із протоколом Codable (рисунок 5.3).

За допомогою експериментів та потужних інструментів для налагодження програмного коду вдалося досягти, без перебільшення, абсолютного точного повторення поведінки декодування об'єктів із використанням протоколу Codable.

Крім того, додатково було реалізовано можливість десеріалізації масиву невизначеної структури та вкладеності (рисунок 5.4). Така можливість може бути корисною, якщо справа йде про якісь специфічні набори даних (наприклад, фігури на шаховій дошці), структуру яких можна передати окремо по протоколу, заздалегідь узгодженому між клієнтом та веб-сервером.

```swift
extension DecodableMO where Self: NSManagedObject {

    static func getObjects(from decoder: Decoder) throws -> [Self] {
        let context = try decoder.getContext()
        var objects = [Self]()
        var container = try decoder.unkeyedContainer()
        while !container.isAtEnd {
            var encodedObject = try self.getEncodedObject(from: &container)
            objects.append(try self.getObject(with: &encodedObject, in: context))
        }
        return objects
    }

    static func getObject(with info: inout KeyedDecodingContainer<MOCodingKeys>, in context: NSManagedObjectContext) throws -> Self {
        let request = Self.MOFetchRequest(with: try self.getPredicate(for: info))
        if let existing = try? context.fetch(request).first, let existingObject = existing {
            return try existingObject.updated(from: info)
        } else {
            return try Self.init(encodedObject: info, context: context)
        }
    }

    static func getEncodedObject(from container: inout UnkeyedDecodingContainer) throws -> KeyedDecodingContainer<MOCodingKeys> {
       return try container.nestedContainer(keyedBy: MOCodingKeys.self)
    }

}
```

Рисунок 5.3 – Реалізація логіки декодування об’єкту

Створені класи-обгортки разом із реалізованою логікою повністю покривають увесь перелік можливих варіантів розташування декодованого об’єкту у JSON-відповіді від веб-серверу.

5.2 Деталі роботи системи

Не дивлячись на наявність вирішуваної проблеми, вбудований протокол Codable має дуже потужний функціонал та значно зменшує кількість коду та зусиль, котрі необхідно прикласти для отримання об’єкту потрібного класу із відповідної JSON-відповіді.

Однією з вимог до розроблюваної системи є відсутність відмінностей у поведінці із вище зазначеним протоколом, а саме, на вході система повинна отримувати JSON-об'єкт, а на виході — повертати готовий до використання об'єкт вказаного класу. Крім цього, головною вимогою до системи є саме уникнення при цьому будь-якої можливості дублювання об'єктів у базі даних.

```
extension DecodableMO where Self: NSManagedObject {

    static func getNestedArrays(from decoder: Decoder) throws -> NSArray {
        var result = NSMutableArray()
        var container = try decoder.unkeyedContainer()
        let context = try decoder.getContext()
        try self.getNestedArrays(from: &container, to: &result, in: context)
        return result

    }

    static private func getNestedArrays(from container: inout UnkeyedDecodingContainer, to array: inout NSMutableArray, in context:
        NSManagedObjectContext) throws -> Void {
        while !container.isAtEnd {
            // Nested Array Exists
            if var nestedArraysContainer = try? container.nestedUnkeyedContainer() {
                var nestedArray = NSMutableArray()
                try self.getNestedArrays(from: &nestedArraysContainer, to: &nestedArray, in: context)
                array.add(nestedArray)
            }
            // It's just a single object
            else {
                var encodedObject = try self.getEncodedObject(from: &container)
                array.add(try self.getObject(with: &encodedObject, in: context))
            }
        }
    }

}
```

Рисунок 5.4 – Реалізація логіки декодування масиву невизначеної структури та вкладеності об'єктів вказаного типу

Нажаль, жоден із розглядуваних у ході дипломної роботи варіантів розв'язку поставленої задачі не дав змоги перевикористати представлену компанією Apple обробку класів, котрі реалізують протокол Codable, оскільки, під час аналізу кожного із них, так і не знайшлося рішення, за допомогою котрого вдалося б обійти неявний примусовий виклик конструктора відповідного класу перед безпосередньою перевіркою наявності такого об'єкту у базі даних, що і породжує проблему дублювання об'єктів.

Описана вище проблема означає лише те, що, для задоволення усіх поставлених до розроблюваної системи вимог, необхідно написати власний протокол із ідентичною поведінкою, при цьому котрий би вирішував головну вимогу, поставлену розроблюваній системі.

Перед безпосереднім написанням такого протоколу необхідно було, для початку, проаналізувати роботу системних класів із протоколом Codable. Схема такої взаємодії зображена на рисунку 5.5.

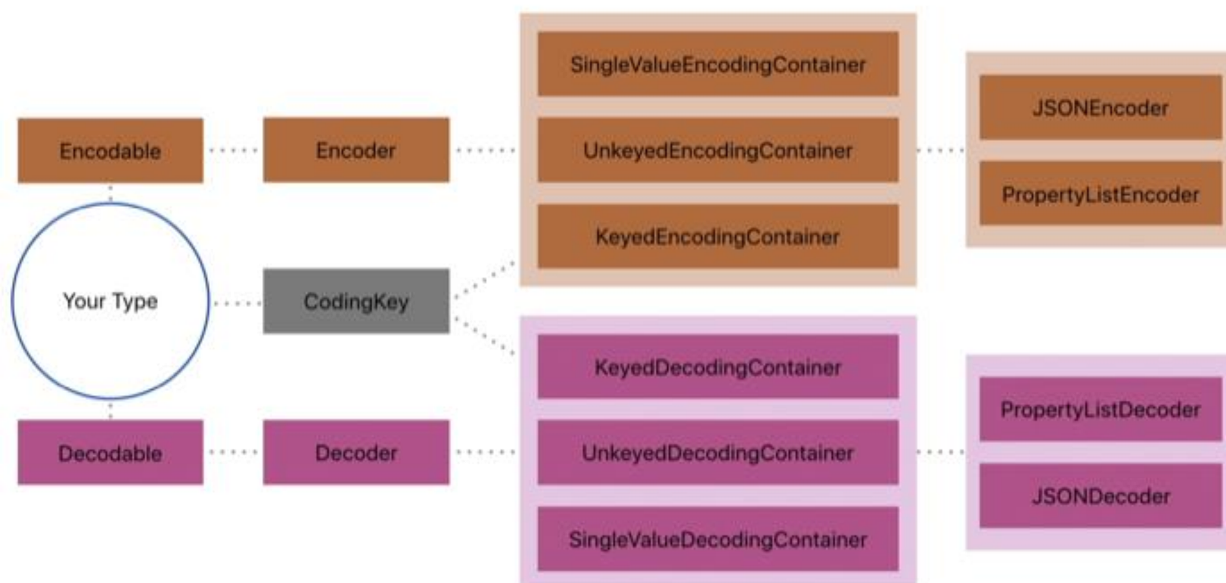

Рисунок 5.5 – Схема взаємодії протоколу Codable із системними класами

У верхній частині наведеної вище схеми зображено частину процесу серіалізації даних, а саме, перетворення об’єкту класу, котрий реалізує протокол Codable, у відповідний йому JSON-об’єкт для подальшого його відправлення на веб-сервер. Варто зауважити, що хоч під час цієї процедури і не виникає жодних проблем, проте новий протокол, котрий буде містити розроблювана система, обов’язково повинен підтримувати такий сценарій взаємодії задля уникнення будь-яких обмежень при його використанні.

Проблемна частина взаємодії системних класів зображена саме на нижній частині представленої схеми. Якщо точніше, проблемною частиною є зв'язок “Decodable – Your Type”, оскільки саме на цьому моменті відбувається додавання декодованого об’єкту у базу даних, що, у випадку наявності там такого об’єкту, призводить до його дублювання.

Реалізованим вирішенням такої проблеми є написання власного протоколу та розширення системних класів методами взаємодії із ним, аби вирішити проблемне місце системного протоколу.

Основними класами, з котрими йде взаємодія, як видно зі схеми, зображеної на рисунку 5.5, є: SingleValueDecodingContainer, UnkeyedDecodingContainer та KeyedDecodingContainer,.

Перший, SingleValueDecodingContainer, використовується лише для базових типів та містить у собі лише одне єдине значення – базового типу.

Другий, UnkeyedDecodingContainer, використовується як контейнер, для зберігання у собі масиву об’єктів, котрі, потенційно, можуть бути отриманими за допомогою використання класу KeyedDecodingContainer.

Останній, KeyedDecodingContainer, використовується як контейнер, для зберігання об’єктних пар “ключ-значення” конкретного об’єкту, який пізніше потрапить до бази даних. Для отримання значення із такого контейнеру необхідно передати у якості параметра одне із значень перелічуваного типу CodingKeys відповідного класу.

У ході розширення системних класів методами роботи із новим протоколом необхідно було дослідити структуру об’єктів класів KeyedDecodingContainer та UnkeyedDecodingContainer, оскільки, як було згадано раніше, перевикористати системну реалізацію методів взаємодії із цими класами просто неможливо при вирішенні даної проблеми. Структури об’єктів вказаних класів вдалося дістати за допомогою високопродуктивного зневаджувача LLDB, вони зображені на рисунках 5.6 та 5.7 відповідно.

Оскільки відповідальність за отримання об’єктів із контейнеру UnkeyedDecodingContainer, котрий містить їх у вигляді масиву вкладених контейнерів, тепер повинна лежати на розроблюваній системі, для реалізації відповідного функціоналу було використано те ж публічне API, котре використовується і системою для тієї ж процедури.

Суть принципу обробки зазначеного контейнеру із масивом полягає у проходженні по ньому за допомогою циклу з метою отримання на кожній ітерації

контейнеру KeyedDecodingContainer. Варто зауважити, що використане API побудоване таким чином, що на кожній наступній ітерації неможливо використовувати сусідні контейнери, що, в свою чергу, змушує на кожній ітерації повністю опрацьовувати поточний контейнер аж до отримання кінцевого значення – об’єкту декодованого класу, аби уникнути дублювання об’єктів у випадку, коли один UnkeyedDecodingContainer містить декілька однакових об’єктів (але це, у загальному випадку, можливо лише за умови неправильної роботи веб-серверу).

```
(lldb) po keyedDecodingContainer.allKeys
▿ 3 elements
  - 0 : CodingKeys(stringValue: "name", intValue: nil)
  - 1 : CodingKeys(stringValue: "is_active", intValue: nil)
  - 2 : CodingKeys(stringValue: "id", intValue: nil)

(lldb) po keyedDecodingContainer.codingPath
▿ 2 elements
  - 0 : CodingKeys(stringValue: "results", intValue: nil)
  ▿ 1 : _JSONKey(stringValue: "Index 0", intValue: 0)
    - stringValue : "Index 0"
    ▿ intValue : Optional<Int>
      - some : 0
```

Рисунок 5.6 – Приклад структури контейнеру KeyedDecodingContainer

Одним із найважливіших кроків у побудові системи було написання розширення для системного класу JSONDecoder, об’єкт котрого і використовується для отримання об’єктів класів-сутностей бази даних із відповідних JSON-відповідей веб-серверу. Клас було розширено трьома шаблонними методами, котрі покривають усі можливі варіанти розміщення об’єкту у JSON-відповіді.

Усі три вище зазначених методи мають однакові сигнатури та відрізняються лише типом значення, яке повертається у результаті їх виконання. Особливості мови Swift дозволяють, в залежності від типу змінної у лівій частині виразу, автоматично обрати необхідний з трьох методів, що значно спрощує роботу користувачів із системою.

Для випадків, коли об’єкти декодуваних класів зберігаються у вигляді значення одного із полів іншого об’єкту, було написано розширення для класу KeyedDecodingContainer.

Розширення для вище зазначеного класу має все ті ж три методи, що і реалізоване розширення для класу JSONDecoder, за єдиною відмінністю – другим аргументом у зазначених функціях є значення перелічуваного типу CodingKeys відповідного класу для задання ключа, за яким зберігається десеріалізовуваний об'єкт, оскільки, як було вказано раніше, використовуючи дане розширення ми отримуємо об'єкт саме за його ключем.

```
(lldb) po unkeyedContainer
▿ _JSONUnkeyedDecodingContainer
  ▿ decoder : <_JSONDecoder: 0x60000239bde0>
  ▿ container : 3 elements
    ▿ 0 : 3 elements
      ▿ 0 : 2 elements
        - key : id
        - value : 2
      ▿ 1 : 2 elements
        - key : name
        - value : Abarth
      ▿ 2 : 2 elements
        - key : is_active
        - value : 1
    ▿ 1 : 3 elements
      ▿ 0 : 2 elements
        - key : id
        - value : 3
      ▿ 1 : 2 elements
        - key : name
        - value : Acura
      ▿ 2 : 2 elements
        - key : is_active
        - value : 1
    ▿ 2 : 3 elements
      ▿ 0 : 2 elements
        - key : id
        - value : 4
      ▿ 1 : 2 elements
        - key : name
        - value : Adler
      ▿ 2 : 2 elements
        - key : is_active
        - value : 1
  - codingPath : 0 elements
  - currentIndex : 0
```

Рисунок 5.7 – Приклад структури контейнеру UnkeyedDecodingContainer

Варто зауважити, що все використане у ході дипломної роботи API є публічним та жодним чином не порушує вимоги компанії Apple щодо розроблюваних додатків та фреймворків. Цей факт робить використання фреймворку безпечним та не наражає розробників на неприємності під час публікації додатку, оскільки будь-який недолік, котрий знайдено компанією Apple за допомогою спеціальних статичних та runtime-аналізаторів під час завантаження додатку до онлайн-магазину додатків App Store,

автоматично піддає додаток додатковим перевіркам та може вплинути на репутацію розробника (компанії).

5.3 Гнучкість та незалежність системи

Розроблена система є максимально гнучкою та незалежною. Єдиним потенційним місцем, де із гнучкістю могли б виникнути проблеми – реалізація протоколу DecodableMO зі сторони користувачів фреймворку, але навіть тут вдалося уникнути таких проблем. Оскільки відповідальність за реалізацію протоколу лежить на плечах користувачів системи, вони можуть імплементувати зазначені методи як завгодно, враховуючи, навіть, зовнішні фактори.

Крім того, конструктор та метод оновлення об'єкту є розділеними, на це є дві причини: по-перше, виклик методів із конструктора, у загальному випадку, не є гарним тоном у програмуванні; по-друге, це збільшує гнучкість системи, оскільки під час оновлення об'єкту можна виконати якусь додаткову логіку, або окремо змінити поля об'єкту, про які веб-сервер, взагалі, не знає, і так далі.

Розроблене рішення представлене у вигляді динамічного фреймворку, а тому із легкістю може бути перевикористаним у необмеженій кількості додатків шляхом найпростішого підключення його до проекту. Вказане представлення системи дозволяє замінити непотрібне копіювання файлів, або, навіть, шматочків коду, на декілька натисків лівої кнопки миші.

Щодо незалежності – система побудована на шаблонних класах та функціях, що робить її абсолютно незалежною від моделі об'єкту, який декодується. Крім цього, система є і контекстно-незалежною, оскільки об'єкти, отримані в результаті десеріалізації, будуть знаходитися у контексті, який було передано користувачем перед початком процесу. Це, в свою чергу, означає, що система повністю готова до використання із мультиконтекстних додатків.

Основний протокол системи – DecodableMO – наслідувано від системного протоколу Decodable. Зроблено це з однією метою – заради поліморфізму. Тепер, у випадку, коли користувач напише розширення для вбудованого фреймворку

Decodable, таке розширення буде доступним і для розробленого протоколу DecodableMO (інакше потрібно було б продублювати код і для розробленого протоколу). Це ж рішення допомагає опрацьовувати об’єкти, які слідують розробленому протоколу у тих же шаблонних функціях, що і об’єкти вбудованого протоколу.

Висновки до розділу 5

У даному розділі наведено блок-схему та алгоритм роботи розробленої системи, описано деталі її програмної реалізації і аргументовано її гнучкість та незалежність від будь-яких факторів.

6 ТЕСТУВАННЯ РОЗРОБЛЮВАНОГО РІШЕННЯ

Тестування програмних продуктів є невід'ємною частиною та дуже важливим етапом у процесі їх створення. Покриття написаного коду тестами дозволяє ще на етапі розробки виявляти недоліки розроблюваної системи та допомагає уникати непередбачених помилок під час рефакторингу її модулів.

Розроблена система являє собою динамічний фреймворк, який не має графічного інтерфейсу, а лише містить у собі логіку, котра розв'язує описану у вступі до роботи проблему, тому можливість та необхідність написання тестів для графічного інтерфейсу відсутня. Натомість, для тестування вище зазначеної логіки була потреба у написанні відповідних юніт-тестів, для цього використовувалася методологія Test-driven development (TDD).

6.1 Методологія Test-driven development

Методологія Test-driven development — це процес розробки програмного забезпечення, який спирається на повторення дуже короткого циклу розробки: вимоги до розроблюваної системи перетворюються на дуже специфічні тестові випадки, після чого програмне забезпечення покращується, щоб пройти тільки нові тести. Це протиставляється розробці програмного забезпечення, яке дозволяє додавати програмне забезпечення, що не відповідає вимогам [16]. На рисунку 6.1 зображена спрощена схема принципу методології.

Розробка за вказаною методологією ще до початку реалізації нового функціоналу спонукає програмістів до написання юніт-тестів, котрі встановлюють вимоги та поведінку коду. Самі тести містять найрізноманітніші порівняння для перевірки задоволення вищезазначених критеріїв. Якщо тестований код задовольняє усім поставленим вимогам, говорять, що тест пройдено. Проходження тесту свідчить про те, що написаний код має очікувану на етапі проектування поведінку. Дуже часто, для тестування, інженери користуються відповідними бібліотеками, котрі надають

зручний функціонал для створення та автоматизації запуску тестів. У дійсності ж, юніт-тести пишуться для складних та заплутаних ділянок коду. До таких ділянок можна віднести код, котрий дуже часто змінюється, або ж код, від результату роботи якого залежить робота дуже великої кількості інших модулів [17].

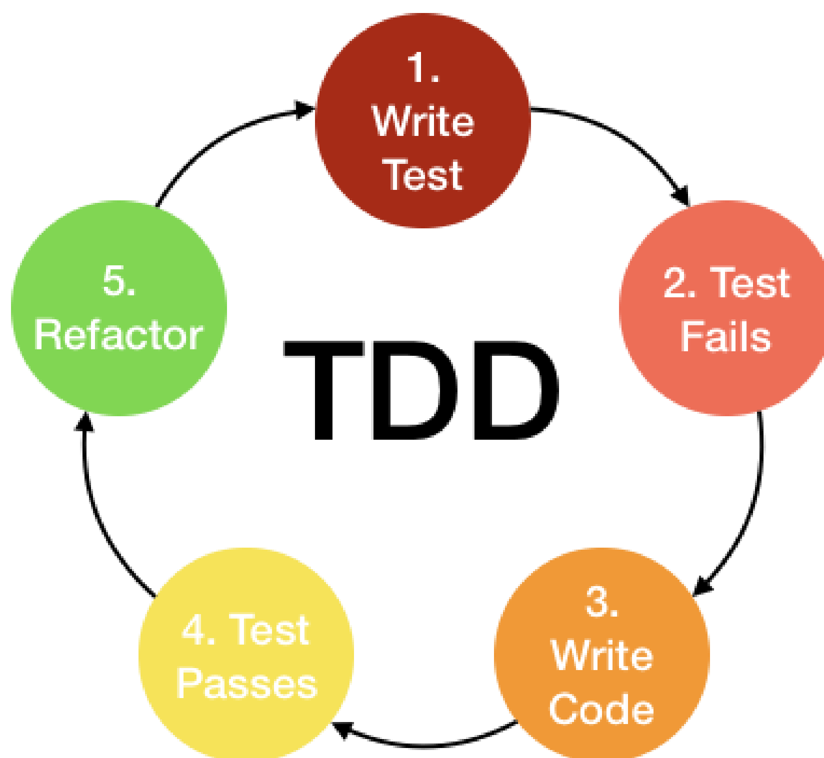

Рисунок 6.1 — Спрощена схема принципу методології TDD

Використана методологія не тільки надає можливість перевірки коду на відповідність різноманітним критеріям, а і безпосередньо впливає на архітектуру розроблюваної системи. З огляду на написані тести, інженери можуть мати чітку уяву про те, який саме функціонал повинна мати система. Таким чином, деталі інтерфейсу можуть з'являтися ще задовго до кінцевої реалізації рішення.

Варто зауважити, що процес тестування є точно таким же важливим, як і процес написання реалізації для спроектованої системи, тому для тестового коду необхідно дотримуватися точно тих же вимог та стандартів, що і до основного коду [18].

На рисунку 6.2 зображена детальна схема принципу методології TDD. Згідно наведеному принципу, процес розробки програмного продукту поділяється на етапи,

які являють собою процес розробки окремих модулів або функцій. Процес же розробки таких модулів складається з наступних етапів:

— написання тестів: під час розробки за методологією TDD розробка кожного модуля або функції починається із написання тестів, які покривають відповідний функціонал та визначають, чи відповідає поведінка модуля заздалегідь прогнозованій, чи ні. При написанні таких тестів, ще до початку реалізації функціоналу, розробники повинні чітко розуміти вимоги та призначення розроблюваного модуля, аби результат тесту точно показував успішність реалізації відповідного функціоналу;

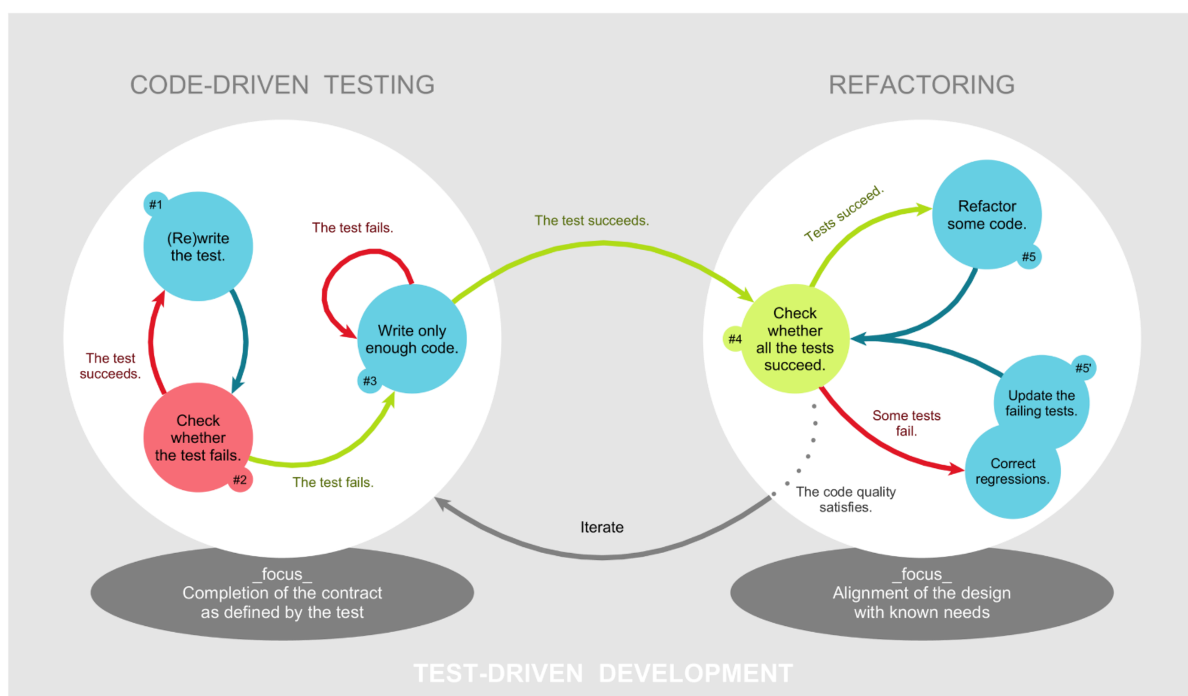

Рисунок 6.2 — Детальна схема принципу методології TDD

— запустити щойно написані тести та переконатися, що результат їх виконання не є успішним: це є підтвердженням того, що тест працює правильно, а його результат підтверджує те, що існуюча поведінка системи з певних, зрозумілих для нас, причин не відповідає очікуваній тестом поведінці. Цей крок надає розробнику впевненості при розробці, оскільки у випадку успішного проходження тесту була б велика ймовірність того, що поведінка модуля, очікувана тестом, із самого початку була неправильною, і у випадку правильної реалізації відповідного

функціоналу це було б неможливо підтвердити, оскільки результат тесту не змінився б;

— написання коду: наступним кроком при розробці за вказаною методологією є реалізація очікуваної поведінки для успішного проходження написаних тестів. Код, написаний на даному кроці хоч і змусить тести змінити свій результат, але аж ніяк не обов'язково буде ідеальним. Проте, цей факт є очікуваним, у наступних кроках написаний код буде удосконалюватися у відповідності до вимог та архітектури розроблюваної системи, при цьому, не змінюючи своєї поведінки;

— повторно запустити написані тести та переконатися в успішності результату: у випадку, коли на даному кроці усі написані тести мають позитивний результат, розробник може бути впевненим, що реалізований ним функціонал має очікувану поведінку;

— проведення рефакторингу: оскільки головною метою етапу написання коду була реалізація поведінки, очікуваної тестом, дотримання стилю та архітектури проекту могло бути опущеним. На цьому етапі, будучи впевненим що тест, цілком точно, написано правильно, саме час удосконалити та "відшліфувати" написаний код таким чином, щоб він точно відповідав усім поставленим перед ним вимогам та чітко вписувався у існуючу архітектуру створюваної системи;

— повторення: усі вище описані кроки необхідно повторювати для кожного окремого тесту модуля. Розмір тесту не повинен бути занадто великим, а сам тест повинен перевіряти поведінку лише однієї функції. При цьому, якщо написаний код із певних причин не задовольняє тест, необхідно встановити причини такої поведінки шляхом налагодження функції та вирішити цей недолік, або ж, за необхідності, відмінити останні зміни за допомогою можливостей системи контролю версій [18].

Юніт-тести призначені для того, щоб протестувати один незалежний модуль. При цьому кількість тестів, призначених для тестування юніту чітко залежить від його функціональності. Головним правилом таких тестів є те, що вони ніяким чином не повинні перетинати кордони свого юніту, використовуючи функціонал сусідніх модулів, або, тим паче, мережеві запити. Пояснюється таке обмеження тим, що воно запобігає впливу роботи інших модулів на час та результати виконання тестів. Адже,

при залежності тесту від зовнішніх факторів ускладнює процес виявлення кореневої причини розбіжностей із очікуваним результатом [19].

Вирішується ж вище зазначена обмеженість шляхом створення так званих фейків та моків, котрі найпростішим чином симулюють очікувану від сусідніх модулів поведінку. Так, наприклад, замість використання допоміжного класу для виконання мережевого запиту, можна створити клас, який реалізує відповідний інтерфейс, але замість мережевого запиту відразу ж повертає деякий кешований результат. Таким чином уникається залежність від реалізації допоміжних класів, якості інтернет-з’єднання, і так далі [20].

6.2 Тестування поведінки системи

Однією з вимог до розроблюваної системи є відсутність різниці поведінки у порівнянні із вбудованим протоколом Codable. Основним призначенням зазначеного протоколу є мінімізація участі розробників у процесі десеріалізації інформації, отриманої від веб-серверу.

На рисунку 6.3 зображена реалізація допоміжних методів, винесених до батьківського для усіх тестів класу, оскільки, як було згадано раніше, вимоги до тестового коду точно такі ж, як і до основного коду. Таке рішення допомогло уникнути дублювання коду та зберегти відведений на тестування системи час.

Для тестування відмінностей поведінки протоколів необхідно було написати тести для десеріалізації отриманих даних, оскільки єдиною очікуваною поведінкою протоколу є успішне декодування отриманих даних.

Реалізація таких тестів зображена на рисунку 6.4. При чому, окрім чотирьох базових випадків розташування десеріалізовуваного об’єкту, які можливі у роботі із протоколом Codable, а саме: у корені JSON-відповіді; як значення, отримуване за деяким ключем із JSON-відповіді, у вигляду масиву об’єктів, які лежать у корені JSON-відповіді; у вигляді масиву об’єктів, які можна отримати за деяким ключем із JSON-відповіді, також було розглянуто два достатньо рідких, але не менш важливих варіанти.

Протокол Codable обмежує користувачів чотирма вище описаними варіантами розміщення об’єкту, що, майже завжди, повністю задовольняє його користувачів.

```
// MARK: - Decoding
// MARK: Just Decoding
func decodeMO<T>(from data: Data) throws -> T where T: DecodableMO {
    return try self.decoder.decodeMO(T.self, from: data)
}

func decodeMO<T>(from data: Data) throws -> [T] where T: DecodableMO {
    return try self.decoder.decodeMO(T.self, from: data)
}

func decode<T>(from data: Data) throws -> T where T: Decodable {
    return try self.decoder.decode(T.self, from: data)
}

// MARK: Decoding with Number of entries in Storage
func numbereddecodeMO<T>(from data: Data) -> (object: T, number: Int) where T: DecodableMO {
    return (try! decodeMO(from: data), getCountInDB())
}

func numbereddecodeMO<T>(from data: Data) -> (objects: [T], number: Int) where T: DecodableMO {
    return (try! decodeMO(from: data), getCountInDB())
}

func numberedDecode<T>(from data: Data) -> (object: T, number: Int) where T: Decodable {
    return (try! decode(from: data), getCountInDB())
}

// MARK: Number of entries in Storage
func getCountInDB() -> Int {
    var count = -1
    let expectation = XCTestExpectation(description: "Getting count in DB")
    self.coreDataManager.saveAllChanges()
    self.coreDataManager.getCarBrands { (carBrands) in
        count = carBrands?.count ?? -1
        expectation.fulfill()
    }
    wait(for: [expectation], timeout: 5.0)
    return count
}
```

Рисунок 6.3 —Допоміжні методи базового тестового класу

Розроблений протокол DecodableMO, крім цього, дозволяє користувачам отримувати декодовані масиви будь-якої структури, при цьому задаючи лише модель об’єктів, які зберігаються у такому масиві. При цьому, неважливо, лежить такий масив у корені JSON-об’єкту, чи його необхідно отримати за деяким з його ключів. Останні два тести, реалізацію яких зображено на рисунку 6.4, тестують саме ці випадки.

6.3 Тестування вирішення проблеми дублювання

Основною і головною вимогою до розроблюваного фреймворку є саме вирішення проблеми дублювання об’єктів, оскільки така задача досі не була розв’язаною. Реалізація відповідних тестів зображена на рисунку 6.5.

```
func test_decodingSingleObject() {
    let encodedObject = ResponseSynthesization.singleObject!
    guard let singleObject: CarBrand = try? self.decodeMO(from: encodedObject) else {
        return XCTFail("DecodableMO-obj should be successfully decoded")
    }
    XCTAssert(singleObject.id == 2, "Unexpected id DecodableMO-obj")
}

func test_decodingArrayOfObjects() {
    let encodedObject = ResponseSynthesization.arrayOfObjects!
    guard let arrayOfObjects: [CarBrand] = try? self.decodeMO(from: encodedObject) else {
        return XCTFail("DecodableMO-objs should be successfully decoded")
    }
    XCTAssert(arrayOfObjects.count == 10, "Not all DecodableMO-objs form response were parsed")
    XCTAssert(arrayOfObjects.first?.id == 2, "Unexpected id of first DecodableMO-obj from Array")
}

func test_decodingSingleObjectAsProperty() {
    let encodedObject = ResponseSynthesization.singleObjectAsProperty!
    guard let carBrandAsProperty: CarBrandAsPropertyResponse = try? self.decode(from: encodedObject) else {
        return XCTFail("Object with DecodableMO-obj as property should be parsed")
    }
    XCTAssert(carBrandAsProperty.result.id == 2, "Unexpected id of DecodableMO-obj")
}

func test_decodingArrayOfObjectsAsProperty() {
    let encodedObject = ResponseSynthesization.arrayOfObjectsAsProperty!
    guard let carBrandsAsProperty: CarBrandsAsPropertyResponse = try? self.decode(from: encodedObject) else {
        return XCTFail("Object with DecodableMO-objs as property should be parsed")
    }
    XCTAssert(carBrandsAsProperty.results.count == carBrandsAsProperty.count, "Not all DecodableMO-objs form response were parsed")
    XCTAssert(carBrandsAsProperty.results.first?.id == 2, "Unexpected id of first DecodableMO-obj from Array")
}

func test_decodingNestedArrays() {
    let encodedObject = ResponseSynthesization.arrayOfArraysOfObjects!
    guard let _: NSArray = try? self.decoder.decodeMO(CarBrand.self, from: encodedObject) else {
        return XCTFail("Nested arrays with DecodableMO-objs should be parsed")
    }
}

func test_decodingComplexNestedArraysAsProperty() {
    let encodedObject = ResponseSynthesization.complexArrayOfArraysAsProperty!
    guard let _ = try? self.decoder.decode(CarBrandsComplexArrayAsPropertyResponse.self, from: encodedObject) else {
        return XCTFail("Nested arrays with DecodableMO-objs should be parsed")
    }
}
```

Рисунок 6.4 — Тестування поведінки розробленого протоколу DecodableMO

Логіка таких тестів достатньо проста: для усіх вище зазначених варіантів розміщення об’єктів у базі даних потрібно двічі поспіль виконати операцію декодування та перевірити кількість об’єктів, котрі лежать у базі даних. Якщо після

повторної десеріалізації кількість об’єктів у базі не змінилася — система спрацювала правильно, а дублювання об’єкту у базі даних для такого випадку є неможливим.

6.4 Тестування контекстної незалежності

Одним із головних принципів незалежності системи, яка використовує фреймворк CoreData, є її контекстна незалежність. Тестування залежності такого типу зображено на рисунку 6.6.

```
func test_duplicatingSingleObject() {
    let encodedObject = ResponseSynthesization.singleObject!
    let firstResponse:(object: CarBrand, number: Int) = self.numbereddecodeMO(from: encodedObject)
    let secondResponse:(object: CarBrand, number: Int) = self.numbereddecodeMO(from: encodedObject)
    XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
    XCTAssert(firstResponse.object === secondResponse.object, "DecodableMO-obj shouldn't be duplicated")
}

func test_duplicatingArrayOfObjects() {
    let encodedObject = ResponseSynthesization.arrayOfObjects!
    let firstResponse:(objects: [CarBrand], number: Int) = self.numbereddecodeMO(from: encodedObject)
    let secondResponse:(objects: [CarBrand], number: Int) = self.numbereddecodeMO(from: encodedObject)
    XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
    XCTAssert(firstResponse.objects.first! == secondResponse.objects.first!, "DecodableMO-obj shouldn't be duplicated")
}

func test_duplicatingSingleObjectAsProperty() {
    let encodedObject = ResponseSynthesization.singleObjectAsProperty!
    let firstResponse:(object: CarBrandAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
    let secondResponse:(object: CarBrandAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
    XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
    XCTAssert(firstResponse.object.result == secondResponse.object.result, "DecodableMO-obj shouldn't be duplicated")
}

func test_duplicatingArrayOfObjectsAsProperty() {
    let encodedObject = ResponseSynthesization.arrayOfObjectsAsProperty!
    let firstResponse:(object: CarBrandsAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
    let secondResponse:(object: CarBrandsAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
    XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
    XCTAssert(firstResponse.object.results.first == secondResponse.object.results.first, "DecodableMO-obj shouldn't be duplicated")
}
```

Рисунок 6.5 — Тестування вирішення проблеми дублювання об’єктів

Як відомо, будь-який об’єкт, котрий є сутністю бази даних, створеної за допомогою фреймворку CoreData, створюється на основі певного контексту. Такий підхід у системі управління базами даних не новий та достатньо популярний, особливо для розробників програмного забезпечення на основі платформи .NET Framework.

Розроблена система є максимально гнучкою та у результаті своєї роботи повертає користувачеві об’єкт, створений на контексті, котрий було передано у

конструктор декодеру, який було використано у цьому процесі. У випадку, якщо декодер було створено без використання контексту, система згенерує виключну ситуацію, котра повідомить розробника про такий недолік.

```
func test_contextIndependency() {
    self.addObjectToDatabase()

    // Contexts
    let mainContext = self.coreDataManager.mainContext
    let privateContext = self.coreDataManager.networkingContext

    // Decoders
    let mainQueueDecoder = JSONDecoder(context: mainContext)
    let privateQueueDecoder = JSONDecoder(context: privateContext)

    let encodedObject = ResponseSynthesization.singleObject!
    let objectOnMainQueue: CarBrand = try! mainQueueDecoder.decodeMO(CarBrand.self, from: encodedObject)
    let objectOnPrivateQueue: CarBrand = try! privateQueueDecoder.decodeMO(CarBrand.self, from: encodedObject)

    XCTAssert(objectOnMainQueue.managedObjectContext == mainContext, "Object should be on main context")
    XCTAssert(objectOnPrivateQueue.managedObjectContext == privateContext, "Object should be on private context")

    XCTAssert(objectOnMainQueue.id == 2, "id should be equals to 2")
    XCTAssert(objectOnPrivateQueue.id == 2, "id should be equals to 2")

    objectOnMainQueue.id = 7
    XCTAssert(objectOnMainQueue.id == 7, "id should be equals to 7")
    XCTAssert(objectOnPrivateQueue.id == 2, "id should be equals to 2")
}

func addObjectToDatabase() {
    let countBefore = self.getCountInDB()
    XCTAssert(countBefore == 0, "Database should be empty")

    let context = self.coreDataManager.mainContext
    let decoder = JSONDecoder(context: context)
    let encodedObject = ResponseSynthesization.singleObject!
    let _: CarBrand = try! decoder.decodeMO(CarBrand.self, from: encodedObject)
    self.coreDataManager.saveAllChanges()

    let countAfter = self.getCountInDB()
    XCTAssert(countAfter == 1, "New object should be added to database")
}
```

Рисунок 6.6 — Тестування контекстної незалежності розробленого рішення

Як було зазначено раніше, контекст, на якому створюються декодовані об’єкти, отримується із декодера, котрий зберігає його на протязі свого життя.

6.5 Тестування впливу на продуктивність

Одне з головних питань до систем такого типу є питання ефективності використання системних ресурсів. Не секрет, що завдяки додатковій логіці, яка відповідає за перевірку наявності декодованого об’єкту у базі даних, система працює

повільніше, аніж без використання зазначеної логіки, це було заздалегідь очевидним фактом. Питання полягає саме у тому, наскільки змінилася швидкість роботи додатку із використанням розробленої системи.

Для отримання відповіді на це питання було написано ряд тестів на продуктивність, приклад якого зображено на рисунку 6.7. Дослідження підтвердили той очевидний факт, що додана із ціллю уникнення дублювання логіка сповільнила роботу системи із фреймворком CoreData, проте різниця у часі є настільки незначною, що, зважаючи на важливість розв'язуваної проблеми, нею можна знехтувати.

```
// MARK: SingleObject
func test_singleObject() {
    print("=========== SingleObject ===========")
    let encodedObject = ResponseSynthesization.singleObject!
    self.measure {
        let _: CarBrand = try! self.decodeMO(from: encodedObject)
    }
}

func test_defaultSingleObject() {
    print("=========== DefaultSingleObject ===========")
    let encodedObject = ResponseSynthesization.singleObject!
    self.measure {
        let _: DefaultCarBrand = try! self.decode(from: encodedObject)
    }
}
```

Рисунок 6.7 — Приклад тестування розробленої системи на продуктивність

Тести показали, різниця при декодуванні об'єкту, який лежить у корені JSON-відповіді становить 0,000104 секунди (0,000136 секунди проти 0,000032 секунди) та 0,0052 (0,0065 проти 0,0013) для масиву із десяти об'єктів, котрий отримано за відповідним ключем отриманого JSON-об'єкту.

Тут важливо зауважити, що зазначена вище різниця у часі не є значною для людського ока, крім того, вона повністю нівелюється у випадку використання мультиконтекстного підходу до управління базою даних, оскільки для десеріалізації отримуваних даних, як правило, використовуються приватні контексти, які не впливають на роботу головного потоку програмного продукту, тим самим, жодним

чином не сповільнюють роботу користувацького інтерфейсу, що робить користування додатком значно гнучкішим та комфортнішим.

Крім того, варто зазначити, що розмір моделі десеріалізовуваного об’єкту ніяк не впливає на швидкість описуваного процесу. Єдиним чинником, який може впливати на швидкість зазначеного процесу є складність предикату, котрий використовується для перевірки наявності декодовуваного об’єкту в базі даних, проте, зазвичай, такий предикат є дуже простим та являє собою вираз виду “id == 1”.

Висновки до розділу 6

У даному розділі описано основну методологію, котра використовувалася під час тестування розроблюваної системи, а також наведено приклади тестів, які були написані упродовж усієї реалізації рішення поставленої задачі. Кожен із написаних тестів мав неабияку цінність під час рефакторингу системи та має стабільно задовільний результат при тестуванні кінцевого вигляду її вихідного коду.

7 РОБОТА КОРИСТУВАЧА ІЗ РОЗРОБЛЕНОЮ СИСТЕМОЮ

7.1 Системні вимоги

Оскільки безпосередніми користувачами розробленої системи є розробники програмного забезпечення для платформ iOS та macOS, єдиною системною вимогою є наявність операційної системи macOS із встановленим середовищем розробки Xcode версії 9.0, або вище.

7.2 Робота розробника зі створеним фреймворком

Для використання розробленого фреймворку розробник повинен підключити його, як і усі інші бібліотеки, використовуючи налаштування створеного проекту.

Одними з основних вимог до розробленого фреймворку були гарна документованість та відсутність різниці у поведінці у порівнянні із вбудованим функціоналом, тому користуватися ним точно так само просто, як і протоколом Codable.

Перш за все, при створенні класу для кожної із сутності бази даних, розробник повинен реалізувати для нього два методи та конструктор протоколу DecodableMO (рисунок 7.1).

Конструктор протоколу буде викликатися у випадках, якщо десеріалізований об’єкт відсутній у базі даних, тобто, такий об’єкт необхідно створити.

Метод updated буде викликатися у випадках, коли десеріалізований об’єкт вже було раніше додано до бази, тобто, його необхідно всього лише оновити актуальною інформацією.

Метод getPredicate буде викликатися кожного разу, коли буде відбуватися перевірка на дублювання. Він повинен повернути об’єкт класу NSPredicate, котрий

буде використано для однозначної ідентифікації присутності декодованого об’єкту у базі даних.

```
/// A type that can decode itself from an external representation.
public protocol DecodableMO : NSManagedObject, Decodable {

    /// A type that is used as a key for encoding and decoding.
    associatedtype MOCodingKeys : CodingKey

    /// Just a typealias to reduce the length of the type
    public typealias EncodedObjectType = KeyedDecodingContainer<Self.MOCodingKeys>

    /// Updates and returns an existing NSManagedObject
    /// instance with given in KeyedDecodingContainer info.
    ///
    /// This method throws an error if reading from the encoded object
    /// fails, or if the data read is corrupted or otherwise invalid.
    ///
    /// An example of implementation:
    /// ````
    /// public func updated(from encodedObject: KeyedDecodingContainer<MOClass.CodingKeys>) throws -> Self {
    ///     self.intProperty = try encodedObject.decode(Int.self, forKey: .intPropertyKey)
    ///     self.customMOProperty = try encodedObject.decodeMO(CustomMOClass.self, forKey: .customMOPropertyKey)
    ///     return self
    /// }
    /// ````
    ///
    /// - Parameter encodedObject: Keyed container with encoded info.
    public func updated(from encodedObject: Self.EncodedObjectType) throws -> Self

    /// Creates a new NSManagedObject instance by decoding
    /// info from the given KeyedDecodingContainer.
    ///
    /// This initializer throws an error if reading from the encoded object
    /// fails, or if the data read is corrupted or otherwise invalid.
    ///
    /// An example of implementation:
    /// ````
    /// required convenience public init(encodedObject: KeyedDecodingContainer<MOClass.CodingKeys>, context: NSManagedObjectContext) throws {
    ///     self.init(entity: MOClass.getEntityDescription(in: context), insertInto: context)
    ///     self.intProperty = try encodedObject.decode(Int.self, forKey: .intPropertyKey)
    ///     self.customMOProperty = try encodedObject.decodeMO(CustomMOClass.self, forKey: .customMOPropertyKey)
    /// }
    /// ````
    ///
    /// - Parameters:
    ///   - encodedObject: Keyed container with encoded info.
    ///   - context: The context that will be used for creating the object.
    public init(encodedObject: Self.EncodedObjectType, context: NSManagedObjectContext) throws

    /// Returns a predicate to determine if this object already exists
    /// in the database.
    ///
    /// This method throws an error if reading from the encoded object
    /// fails, or if the data read is corrupted or otherwise invalid.
    ///
    /// An example of implementation:
    /// ````
    /// public static func getPredicate(for encodedObject: KeyedDecodingContainer<MOClass.CodingKeys>) throws -> NSPredicate {
    ///     let key = CodingKeys.intPropertyKey.stringValue
    ///     let value = try encodedObject.decode(Int.self, forKey: .intPropertyKey)
    ///     return NSPredicate(format: "%K == %d", key, value)
    /// }
    /// ````
    ///
    /// - Parameter encodedObject: Keyed container with encoded info.
    public static func getPredicate(for encodedObject: Self.EncodedObjectType) throws -> NSPredicate
}
```

Рисунок 7.1 – Протокол DecodableMO розробленого фреймворку

На рисунку 7.2 зображені методи, котрими було розширено зазначений раніше протокол DecodableMO. Ці методи використовуються безпосередньо зсередини розробленого фреймворку, але, зважаючи на їх користь, було вирішено зробити їх

публічними, адже під час розробки програмних продуктів іноді буває необхідність реалізовувати такі методи власноруч.

```
extension DecodableMO {

    /// Returns a description of search criteria used to retrieve data from a persistent store.
    ///
    /// - Returns: A description of search criteria used to retrieve data from a persistent store.
    public static func MOFetchRequest() -> NSFetchRequest<Self>

    /// Returns a predicate description of search criteria used
    /// to retrieve data from a persistent store.
    ///
    /// - Parameter predicate: The predicate that will be contained in the query
    /// - Returns: A predicate description of search criteria used to retrieve data from a persistent store.
    public static func MOFetchRequest(with predicate: NSPredicate) -> NSFetchRequest<Self>

    /// Returns the entity associated with the specified managed object
    /// context's persistent store coordinator.
    ///
    /// - Parameter context: The managed object context to use. Must not be nil.
    /// - Returns: The entity associated with context's persistent store coordinator.
    public static func getEntityDescription(in context: NSManagedObjectContext) -> NSEntityDescription
}
```

Рисунок 7.2 – Розширення протоколу DecodableMO

Тепер, реалізувавши необхідні для роботи з протоколом методи, перейдемо до безпосереднього процесу десеріалізації об’єктів. На рисунку 7.3 зображено розширення системного класу JSONDecoder методами для десеріалізації об’єктів, які реалізують протокол DecodableMO та конструктором, котрий створює об’єкт вказаного класу із використанням екземпляра класу NSManagedObjectContext, отриманим у якості параметра.

Розширення містить три методи, котрі будуть використовуватися в одному з трьох випадків: якщо об’єкт десеріалізовуваного класу лежить у корені отриманого JSON-об’єкту, якщо коренем JSON-об’єкту є масив (чітко визначеної структури, одиничної вкладеності) об’єктів десеріалізовуваного класу, або ж якщо коренем JSON-об’єкту є масив невизначеної структури із об’єктами десеріалізовуваного класу.

Для випадку, коли об’єкт десеріалізовуваного класу є однією з властивостей іншого об’єкту, необхідно використовувати розширення для шаблонної системної структури KeyedDecodingContainer.

Як і у випадку розширення для класу JSONDecoder, воно містить три методи, котрі потрібно використовувати в одному з наступних випадків: якщо об'єкт десеріалізовуваного класу лежить у корені JSON-об'єкту, отриманого за вказаним ключем; якщо коренем JSON-об'єкту за вказаним ключем є масив (чітко визначеної структури, одиничної вкладеності) об'єктів десеріалізовуваного класу, або ж якщо коренем JSON-об'єкту за вказаним ключем є масив невизначеної структури із об'єктами десеріалізовуваного класу.

```
extension JSONDecoder {

    /// Decodes a top-level value of the given type from the given JSON representation.
    ///
    /// - parameter type: The type of the value to decode.
    /// - parameter data: The data to decode from.
    /// - returns: A value of the requested type.
    /// - throws: `DecodingError.dataCorrupted` if values requested from the payload are corrupted, or if the given data is not valid JSON.
    /// - throws: An error if any value throws an error during decoding.
    public func decodeMO<T>(_ type: T.Type, from data: Data) throws -> T where T : DecodableMO

    /// Decodes a top-level values array of the given type from the given JSON representation.
    ///
    /// - parameter type: The type of the value to decode.
    /// - parameter data: The data to decode from.
    /// - returns: A value of the requested type.
    /// - throws: `DecodingError.dataCorrupted` if values requested from the payload are corrupted, or if the given data is not valid JSON.
    /// - throws: An error if any value throws an error during decoding.
    public func decodeMO<T>(_ type: T.Type, from data: Data) throws -> [T] where T : DecodableMO

    /// Decodes a top-level values undefined structure array of the given type from the given JSON representation.
    ///
    /// - parameter type: The type of the value to decode.
    /// - parameter data: The data to decode from.
    /// - returns: A value of the requested type.
    /// - throws: `DecodingError.dataCorrupted` if values requested from the payload are corrupted, or if the given data is not valid JSON.
    /// - throws: An error if any value throws an error during decoding.
    public func decodeMO<T>(_ type: T.Type, from data: Data) throws -> NSArray where T : DecodableMO

    /// Initializes `self` with given context and default strategies.
    ///
    /// - Parameter context: The context that will be used for creating the objects.
    public convenience init(context: NSManagedObjectContext)
}
```

Рисунок 7.3 – Розширення для системного класу JSONDecoder

Усі вище описані речі та, навіть, приклади реалізації методів протоколу DecodableMO, можна у документації до відповідних методів, просто перейшовши на документацію до розробленого фреймворку (комбінація клавіш Command + ЛКМ).

7.3 Приклади роботи із фреймворком

На рисунку 7.4 наведено приклад реалізації протоколу DecodableMO для класу сутності TestEntity, котра має три властивості: id, name, amount.

Якщо звернути увагу на метод getPredicate, то можна легко здогадатися, що для ідентифікації наявності об'єкту у базі даних буде зроблено перевірку на присутність в ній екземпляру із декодованим значенням властивості id.

На рисунку 7.5 зображено приклад створення та використання екземпляра класу JSONDecoder для десеріалізації об'єктів, класи яких наслідують протокол Decodable або DecodableMO, в тому числі, і для класу вище описаної сутності.

```
@objc(TestEntity)
public class TestEntity: NSManagedObject, DecodableMO {
    public typealias MOCodingKeys = CodingKeys

    @NSManaged public var id: Int64
    @NSManaged public var name: String
    @NSManaged public var amount: Double


    // MARK: DecodableMO
    public enum CodingKeys: String, CodingKey {
        case id, name, amount
    }

    public required convenience init(encodedObject: KeyedDecodingContainer<TestEntity.CodingKeys>, context: NSManagedObjectContext) throws {
        self.init(entity: TestEntity.getEntityDescription(in: context), insertInto: context)
        self.id = try encodedObject.decode(Int64.self, forKey: .id)
        self.name = try encodedObject.decode(String.self, forKey: .name)
        self.amount = try encodedObject.decode(Double.self, forKey: .amount)
    }

    public func updated(from encodedObject: KeyedDecodingContainer<TestEntity.CodingKeys>) throws -> Self {
        self.name = try encodedObject.decode(String.self, forKey: .name)
        self.amount = try encodedObject.decode(Double.self, forKey: .amount)
        return self
    }

    public static func getPredicate(for encodedObject: KeyedDecodingContainer<TestEntity.CodingKeys>) throws -> NSPredicate {
        let key = CodingKeys.id.stringValue
        let value = try encodedObject.decode(Int64.self, forKey: .id)
        return NSPredicate(format: "%K == %d", key, value)
    }

}
```

Рисунок 7.4 – Приклад реалізації протоколу Decodable

Варто зауважити, що запропонований на рисунку 7.5 приклад використання розширень дозволив за допомогою всього лише трьох шаблонних методів описати три з чотирьох можливих варіантів розташування об'єктів у JSON-відповіді від веб-серверу.

Використання шаблонних методів є необхідним у випадках, коли єдиною відмінністю у реалізованих функціях є вхідні параметри, або тип значення, яке повертається.

Так і у випадку, запропонованому на рисунку 7.5, усі три методи є шаблонними, а аргумент такої функції мають реалізовувати відповідні протоколи.

```
class NetworkHandler {

    // MARK: - Properties
    let context: NSManagedObjectContext

    // MARK: - Methods
    public func handleResponse<T: Decodable>(_ data: Data?, _ response: URLResponse?, _ error: Error?, completionHandler: @escaping (T?) -> Void) -> Void {
        guard let data = data else {
            self.handleResponse(response, with: error)
            return completionHandler(nil)
        }
        let decoder = JSONDecoder(context: context)
        completionHandler(try? decoder.decode(T.self, from: data))
    }

    public func handleResponse<T: DecodableMO>(_ data: Data?, _ response: URLResponse?, _ error: Error?, completionHandler: @escaping (T?) -> Void) -> Void {
        guard let data = data else {
            self.handleResponse(response, with: error)
            return completionHandler(nil)
        }
        let decoder = JSONDecoder(context: context)
        completionHandler(try? decoder.decodeMO(T.self, from: data))
    }

    public func handleResponse<T: DecodableMO>(_ data: Data?, _ response: URLResponse?, _ error: Error?, completionHandler: @escaping ([T]?) -> Void) -> Void {
        guard let data = data else {
            self.handleResponse(response, with: error)
            return completionHandler(nil)
        }
        let decoder = JSONDecoder(context: context)
        completionHandler(try? decoder.decodeMO(T.self, from: data))
    }

    // MARK: Error handling
    private func handleResponse(_ response: URLResponse?, with error: Error?) -> Void {
        // *** Here should be your error handling logic *** \\
    }

    // MARK: - Constructor
    init(context: NSManagedObjectContext) {
        self.context = context
    }

}
```

Рисунок 7.5 – Приклад використання розширень для системного класу JSONDecoder

Цих шаблонних методів, у більшості випадків, буде достатньо для того успішного декодування об’єктів будь-якого класу, котрий реалізує один з двох протоколів – Decodable або DecodableMO уникаючи дублювання коду при створенні такого обробника на кожну із створених сутностей бази даних.

Висновки до розділу 7

У даному розділі описано роботу користувача із розробленою системою, наведено конкретні приклади у вигляді моделей-сутностей бази даних, а також реалізації необхідних для роботи протоколу методів.

ВИСНОВКИ

У ході дипломної роботи було розроблено динамічний фреймворк CodableMO, котрий вирішує проблему взаємодії вбудованої бібліотеки CoreData разом із протоколом Codable, а саме, містить у собі оптимальний розв'язок задачі уникнення дублювання об'єктів у базах даних.

У ході роботи було проаналізовано поставлену задачу, виявлено та синтезовано вирішення потенційної проблеми, внаслідок якої і з'явилася розв'язувана задача. Система має детальну документацію до кожного з її публічних методів, крім того, документація методів протоколу містить у собі приклади їх реалізації.

Система спроектована таким чином, що вона є незалежною від зовнішніх факторів, має належну гнучкість та поведінку, котра не відрізняється від поведінки взаємодії вбудованих класів із системним протоколом Codable. Усі вище зазначені речі у ході розробки тестувалися великою кількістю юніт-тестів, кожен із яких має стабільно задовільний результат при тестуванні кінцевого вигляду вихідного коду системи.

Кінцевими користувачами розробленої системи є сторонні розробники, котрі, в силу поставлених перед ними задач, стикнулися із вирішеною проблемою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mark D. Beginning iPhone Development: Exploring the iOS SDK / D. Mark, J. Nutting, K. Topley., 2014. – 808 с. – (Apress).
2. Нахавандипур В. iOS. Приемы программирования / Вандад Нахавандипур., 2014. – 832 с. – (Бестселлеры O'Reilly).
3. Кочан С. Программирование на Objective-C / Стивен Кочан., 2014. – 550 с. – (6-е издание).
4. Усов В. Swift. Основы разработки приложений под iOS и macOS / Василий Усов., 2016. – 448 с. – (4-е издание). – (Библиотека программиста).
5. Manning J. Learning Swift / J. Manning, P. Buttfield-Addison, T. Nugent., 2018. – 378 с. – (3rd edition).
6. Vilmart F. Hands-On Design Patterns with Swift / F. Vilmart, G. Scalzo, S. De Simone., 2018. – 414 с. – (1st edition).
7. Wals D. Mastering iOS 12 Programming / Donny Wals., 2018. – 750 с. – (3rd edition).
8. Zarra M. Core Data in Objective-C: Data Storage and Management for iOS and OS X / Marcus Zarra., 2016. – 240 с. – (3rd edition).
9. Douglas A. Core Data by Tutorials Third Edition: iOS 10 and Swift 3 edition / A. Douglas, S. Mora, M. Morey., 2016. – 250 с. – (3rd edition).
10. Privat M. Pro iOS Persistence: Using Core Data / M. Privat, R. Warner., 2014. – 388 с. – (1st edition).
11. Roadley T. Learning Core Data for iOS: A Hands-On Guide to Building Core Data Applications / Tim Roadley., 2013. – 480 с. – (1st edition). – (Learning).
12. Ganim E. Encoding, Decoding and Serialization in Swift 4 [Електронний ресурс] / Eli Ganim // raywenderlich. – 2017. – Режим доступу до ресурсу: https://www.raywenderlich.com/382-encoding-decoding-and-serialization-in-swift-4.

13. Framework [Електронний ресурс] // Apple Developer Documentation – Режим доступу до ресурсу: https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/Framework.html.

14. Davies S. How to Create a Framework for iOS [Електронний ресурс] / Sam Davies // raywenderlich. – 2014. – Режим доступу до ресурсу: https://www.raywenderlich.com/2430-how-to-create-a-framework-for-ios.

15. Overview of Dynamic Libraries [Електронний ресурс] // Apple Developer Documentation – Режим доступу до ресурсу: https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/100-Articles/OverviewOfDynamicLibraries.html.

16. Hauser D. Test-Driven iOS Development with Swift / Dr. Dominik Hauser., 2016. – 218 с.

17. Бек К. Экстремальное программирование. Разработка через тестирование / Кент Бек., 2016. – 224 с. – (Библиотека программиста).

18. Greene J. iOS Test-Driven Development by Tutorials / J. Greene, M. Katz., 2019. – 447 с. – (1st edition).

19. Hamill P. Unit Test Frameworks / Paul Hamill., 2004. – 304 с. – (1st edition).

20. Freeman S. Growing Object-Oriented Software, Guided by Tests / S. Freeman, N. Pryce., 2009. – 384 с. – (1st edition).

ДОДАТОК А

Система уніфікації методів доступу до баз даних

Специфікація

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТР5281_19Б

Аркушів 2

Київ 2019

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ТР5281_19Б	Записка.docx	Пояснювальна записка до дипломної роботи
Компоненти		
УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ТР5281_19Б 12-1	CodableMO.xcodeproj	Вихідний код розробленого фреймворку
УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ТР5281_19Б 12-2	CDMultipleContexts .xcodeproj	Приклад використання фреймворку

ДОДАТОК Б

Система уніфікації методів доступу до баз даних

Текст програми

УКР.НТУУ”КПІ”_ТЕФ_АПЕПС_ТР5281_19Б

Аркушів 3

Київ 2019

Реалізація логіки декодування об’єкту:

```
import CoreData

extension DecodableMO where Self: NSManagedObject {

    static func getNestedArrays(from decoder: Decoder) throws -> NSArray {
        var result = NSMutableArray()
        var container = try decoder.unkeyedContainer()
        let context = try decoder.getContext()
        try self.getNestedArrays(from: &container, to: &result, in: context)
        return result
    }

    static private func getNestedArrays(from container: inout UnkeyedDecodingContainer, to array: inout NSMutableArray, in context:
    NSManagedObjectContext) throws -> Void {

        while !container.isAtEnd {
            // Nested Array Exists
            if var nestedArraysContainer = try? container.nestedUnkeyedContainer() {
                var nestedArray = NSMutableArray()
                try self.getNestedArrays(from: &nestedArraysContainer, to: &nestedArray, in: context)
                array.add(nestedArray)
            }
            // It's just a single object
            else {
                var encodedObject = try self.getEncodedObject(from: &container)
                array.add(try self.getObject(with: &encodedObject, in: context))
            }
        }
    }

}
```

Реалізація логіки декодування масиву невизначеної структури та вкладеності об’єктів вказаного типу:

```
import CoreData

extension DecodableMO where Self: NSManagedObject {

    static func getNestedArrays(from decoder: Decoder) throws -> NSArray {
        var result = NSMutableArray()
        var container = try decoder.unkeyedContainer()
        let context = try decoder.getContext()
        try self.getNestedArrays(from: &container, to: &result, in: context)
        return result

    }

    static private func getNestedArrays(from container: inout UnkeyedDecodingContainer, to array: inout NSMutableArray, in context:
    NSManagedObjectContext) throws -> Void {

        while !container.isAtEnd {
            // Nested Array Exists
            if var nestedArraysContainer = try? container.nestedUnkeyedContainer() {
                var nestedArray = NSMutableArray()
                try self.getNestedArrays(from: &nestedArraysContainer, to: &nestedArray, in: context)
                array.add(nestedArray)
            }
            // It's just a single object
            else {
                var encodedObject = try self.getEncodedObject(from: &container)
                array.add(try self.getObject(with: &encodedObject, in: context))
            }
        }
    }

}
```

Тестування вирішення проблеми дублювання:

```
import XCTest
@testable import CDMultipleContexts
@testable import CodableMO

class CDModelsDuplicatingTest: CDMultipleContextsTestCase {

	override func setUp() {
		super.setUp()
	}

	override func tearDown() {
		super.tearDown()
	}

	func test_duplicatingSingleObject() {
		let encodedObject = ResponseSynthesization.singleObject!
		let firstResponse:(object: CarBrand, number: Int) = self.numbereddecodeMO(from: encodedObject)
		let secondResponse:(object: CarBrand, number: Int) = self.numbereddecodeMO(from: encodedObject)
		XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
		XCTAssert(firstResponse.object === secondResponse.object, "DecodableMO-obj shouldn't be duplicated")
	}

	func test_duplicatingArrayOfObjects() {
		let encodedObject = ResponseSynthesization.arrayOfObjects!
		let firstResponse:(objects: [CarBrand], number: Int) = self.numbereddecodeMO(from: encodedObject)
		let secondResponse:(objects: [CarBrand], number: Int) = self.numbereddecodeMO(from: encodedObject)
		XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
		XCTAssert(firstResponse.objects.first! == secondResponse.objects.first!, "DecodableMO-obj shouldn't be duplicated")
	}

	func test_duplicatingSingleObjectAsProperty() {
		let encodedObject = ResponseSynthesization.singleObjectAsProperty!
		let firstResponse:(object: CarBrandAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
		let secondResponse:(object: CarBrandAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
		XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
		XCTAssert(firstResponse.object.result == secondResponse.object.result, "DecodableMO-obj shouldn't be duplicated")
	}

	func test_duplicatingArrayOfObjectsAsProperty() {
		let encodedObject = ResponseSynthesization.arrayOfObjectsAsProperty!
		let firstResponse:(object: CarBrandsAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
		let secondResponse:(object: CarBrandsAsPropertyResponse, number: Int) = self.numberedDecode(from: encodedObject)
		XCTAssert(firstResponse.number == secondResponse.number, "DecodableMO-obj shouldn't be duplicated")
		XCTAssert(firstResponse.object.results.first == secondResponse.object.results.first, "DecodableMO-obj shouldn't be duplicated")
	}

}
```

ДОДАТОК В

Система уніфікації методів доступу до баз даних

Опис програми

УКР.НТУУ"КПІ"_ТЕФ_АПЕПС_ТР5281_19Б

Аркушів 9

Київ 2019

АНОТАЦІЯ

Даний додаток містить опис системи, призначеної для вирішення проблеми дублювання об’єктів у базі даних при використанні фреймворку CoreData із протоколом Codable. Розроблена система має наступні характеристики:

— реалізація рішення у вигляді динамічного фреймворку;

— має оптимальну реалізацію із точки зору впливу на продуктивність;

— має гарну документованість, дотриману стилю документації від Apple;

— не має різниці у поведінці у порівнянні із вбудованим функціоналом – протоколом Codable;

— є максимально гнучкою та незалежною, не має обмежень, з якими б стикнулися сторонні розробники.

ЗМІСТ

1. Загальні відомості 60
2. Функціональне призначення 61
3. Опис логічної структури 62
4. Технічні засоби, що використовуються 63
5. Виклик і завантаження 64
6. Вхідні і вихідні дані 65

ЗАГАЛЬНІ ВІДОМОСТІ

У цьому додатку міститься опис фреймворку, котрий вирішує проблему взаємодії фреймворку CoreData та протоколу Codable. У додатку Б міститься програмний код основних модулів розробленого фреймворку CodableMO.

Розроблена система використовується у додатках, написаних для платформ iOS та macOS. Використання фреймворку при написанні додатків вимагає використання ПК із встановленою операційною системою macOS та середовищем розробки Xcode версії 9.0 (або вище).

Під час розробки системи використовувалася мова програмування Swift, середовище розробки Xcode, а також потужності фреймворку CoreData та протоколу Codable.

ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Розроблена система використовується у випадках необхідності одночасного використання фреймворку CoreData та протоколу Codable, оскільки у такому випадку з'являється проблема дублювання об'єктів у базі даних.

Розроблений фреймворк CodableMO містить у собі оптимальне вирішення зазначеної проблеми та допомагає заощадити стороннім розробникам час на пошук альтернативних рішень поставленої задачі.

ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Для розв'язання поставленої задачі знадобилося створення динамічного фреймворку, котрий містить у собі протокол CodableMO, призначений для вирішення проблеми дублювання об'єктів у базі даних.

Вирішити зазначену проблему вдалося за допомогою класів-обгорток, публічного API для роботи із контейнерами даних, а також потужностей мови Swift, а саме: шаблонних функцій і класів, асоціативних посилань та протоколів.

Рішення представлено у вигляді динамічного фреймворку, а тому може бути з легкістю перевикористаним у будь-якому із можливих додатків, розроблюваних для платформ iOS та macOS.

ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Оскільки стикнутися із вище описаною проблемою можна лише розроблюючи додатки для платформ iOS та macOS, використовувані технічні засоби чітко обмежені точно тими ж інструментами, котрі використовуються і під час розробки додатків для зазначених платформ, а саме: середовище розробки Xcode версії 9.0 (або вище), мова програмування Swift, фреймворк CoreData та протокол Codable.

ВИКЛИК І ЗАВАНТАЖЕННЯ

Для використання реалізованої системи, необхідно, використовуючи налаштування розроблюваного додатку, підключити фреймворк CodableMO.framework до проекту та імпортувати його у всіх файлах проекту, котрі будуть використовувати його функціонал.

Розроблений фреймворк також містить у собі повну документацію по кожному з публічних його методів, включаючи детальні приклади їх використання.

ВХІДНІ І ВИХІДНІ ДАНІ

Вхідними даними для розробленої системи є JSON-об’єкти, отримувані після десеріалізації набору байтів за допомогою статичного методу класу JSONSerialization.

У загальному ж випадку, вхідними даними для системи можуть також бути об’єкти класу Dictionary, котрий і є типом даних мови Swift, котрий використовується для представлення вище зазначених JSON-об’єктів.

Вихідними даними є об’єкт (або ж об’єкти) класу-сутності бази даних, створеної за допомогою фреймворку CoreData, при цьому алгоритм роботи системи виключає можливість його (їх) дублювання.