КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра теорії та технології програмування

Кваліфікаційна робота

на здобуття ступеня бакалавра

За спеціальністю 122 Комп'ютерні науки

на тему:

МОБІЛЬНИЙ ДОДАТОК З ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ ДОПОВНЕНОЇ РЕАЛЬНОСТІ

Виконав студент 4-го курсу

Дмитро ШАБАНОВ _____

(підпис)

Науковий керівник:

кандидат фізико-математичних наук

Андрій КРИВОЛАП _____

(підпис)

Засвідчую, що в роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування

«_____»_____ 2023 р,

протокол №_____

Завідувач кафедри

Микола НІКІТЧЕНКО _____

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 48 сторінок, 9 ілюстрацій, 18 використаних джерел.

Ключові слова: ANDROID, МОБІЛЬНИЙ ДОДАТОК, ДОПОВНЕНА РЕАЛЬНІСТЬ, БАЗИ ДАНИХ, ІНТЕРФЕЙС ПРОГРАМНОГО ПРОДУКТУ, МОБІЛЬНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, FIREBASE.

Об’єктом роботи є процес розробки мобільного додатку з каталогом продукції для магазину меблів із використанням технології доповненої реальності. Предметом роботи є мобільний додаток для ОС Android, що використовує технологію доповненої реальності.

Метою роботи є розробка та створення мобільного додатку з використанням технології доповненої реальності.

Методи розроблення: методи проєктування мобільних додатків, розробка зручного інтерфейсу мобільного додатку, взаємодія з базою даних. Інструменти розроблення: IDE Android Studio Dolphin 2021.3.1, мова програмування Kotlin.

Результати роботи: було проведено аналіз існуючих додатків магазинів меблів, розроблено мобільний додаток для ОС Android, який дозволяє переглянути наявні товари, обрати певну конфігурацію кожного товару і подивитися товар за допомогою технології доповненої реальності.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

IDE – integrated development environment, інтегроване середовище розробки;

SDK - software development kit, засоби розробки для певної платформи;

БД – база даних;

ОС – операційна система;

ВСТУП

Оцінка сучасного стану об’єкта дослідження або розробки. На даний момент технологія доповненої реальності отримує все більшу популярність. Завдяки зростанню потужності сучасних мобільних пристроїв а також наявності інструментів, що дозволяють інтегрувати доповнену реальність у мобільний додаток, щороку з’являється усе більше додатків із цією технологією і вона набуває все більше користувачів. Вона використовується для мобільних ігор, для навігації, для розглядання моделей товарів перед покупкою і для багатьох інших сфер діяльності людини.

Актуальність роботи та підстави для її виконання. Тема допоненої реальності у мобільних пристроях набуває все більшої популярності і актуальності за рахунок спрощення взаємодії з оточуючим світом. Доповнена реальність дозволяє інтегрувати цифрові предмети і явища у оточуючий світ людини, спростивши взаємодію між ними.

Доповнена реальність також є корисною для користувачів інтернет магазинів меблів. Зазвичай, користувачу, щоб побачити, чи буде пасувати та чи інша мебіль до її кімнати, треба або користуватися послугами дизайнера, або шукати зображення використання таких же меблів у кімнатах, що схожі на її кімнату, що не завжди можливо, або ж використовувати уяву і намагатися «на око» прикинути, чи буде мебіль пасувати. Усі ці способи не є надійними, зручними чи швидкими, оскільки вимагають додаткових зусилль і, зазвичай, навіть якщо вони успішно вдаються, дають роздивитися мебіль з невеликої кількості ракурсів.

Якщо ж використовувати доповнену реальність і, за допомогою камери мобільного телефону, дивитися на об’ємну модель меблі у своїй кімнаті, можна роздивитися її з усіх боків, у будь-якому місці кімнати і поставлену саме так, як людина бажає. Це може покращити користувацький досвід та привабити клієнтів.

Мета і завдання роботи. Метою кваліфікаційної роботи є створення мобільного додатку, що використовує ОС Android[1] із функціоналом доповненої реальності. Для досягнення цієї мети було поставлено наступні завдання:

1. Дослідити існуючі додатки магазинів з продажу меблів
2. Базуючись на проведенному дослідженні існуючих додатків, сформулювати вимоги до майбутнього додатку
3. Визначити перелік інструментів та технологій, що використовуватимуться для розробки нашого додатку
4. Спроектувати схему локальної БД
5. Розробити інтерфейс додатку, логіку його роботи та мережевої взаємодії
6. Інтегрувати у додаток технологію доповненої реальності

Об’єкт, методи й засоби дослідження або розроблення. Об’єктом розробки є використання технології доповненої реальності для інтеграції цифрових об’єктів у реальний світ.

Методами розробки є методи проєктування мобільних додатків, розробка інтерфейсу мобільного додатку, взаємодія з базою даних.

Для розробки було використано Android Studio IDE у якості інтегрованого середовища розробки, мова програмування Kotlin[2,3]. Для віддаленого збереження даних було обрано Firebase DataStore[4] та Firebase Storage[5]. Для локального збереження даних була обрана локальна БД із взаємодією через бібліотеку Room[6]. Для створення інтерфейсу користувача було обрано технологію Jetpack Compose[7,8].

РОЗДІЛ 1 ОГЛЯД НАЯВНИХ НА РИНКУ СИСТЕМ

У якості теми додатку було обрано каталог продукції для магазину меблів з можливістю побачити товар у доповненій реальності.

Розглянемо наявні на ринку мобільні додатки зі схожою тематикою та функціоналом:

- IKEA Place – мобільний додаток з функцією перегляду меблів у доповненій реальності відомої мережі магазинів з аналогічною назвою. Він має сучасний інтерфейс, великий асортимент товарів та велику кількість магазинів по усьому світу. Серед недоліків можна зазначити, що він відокремлений від основного додатку IKEA, що ускладнює взаємне використання, а також він недоступний в Україні.
- Amazon – мобільний додаток найбільшого у світі інтернет-магазину. Має широкий перелік товарів та можливість переглянути меблі у доповненій реальності. Серед недоліків є те, що він не орієнтований на український ринок, у ньому відсутній регіон «Україна», а також відсуня українська мова, і через це доставка замовлення кінцевому користувачу може зайняти великий проміжок часу.
- Houzz – мобільний додаток інтернет-магазину з відповідною назвою. Має широкий перелік товарів та можливість переглянути меблі у доповненій реальності. Але, замовити товар можливо лише у Канаду та США
- Taburetka.ua – мобільний додаток українського інтернет-магазину меблів. Загальна ідея схожа на запропонований додаток, проте у ньому немає можливості переглянути товар у доповненій реальності і немає адрес магазинів, у які можна прийти та переглянути обраний товар вживу.

Проаналізувавши перераховані рішення, можна помітити, що ті додатки, у яких присутній функціонал доповненої реальності не орієнтовані

на український ринок, через що їх використання стає недоцільним. А ті, що орієнтовані на український ринок, не мають цього функціоналу. Таким чином ця ніша на даний момент залишається не зайнятою. Розроблений додаток буде покривати цю нішу.

Окрім того, у жодному з перерахованих додатків немає можливості якимось чином налаштувати обрані меблі до вимог користувача. Наприклад, користувач хоче певний товар завширшки у 57 см, бо він має вільний простір саме такого розміру. У звичайних магазинах усі товари мають заздалегідь вказані розміри, які не можуть змінюватись. Можливість вибору свого значення у певному діапазіоні, наприклад від 50 см до 120 см, дозволить обрати саме такий товар, який потрібен користувачу, завдяки чому можна запобігти втрату користувачів, яким потрібен товар зі специфічними властивостями.

У деяких із перерахованих додатків немає можливості переглянути місцезнаходження магазинів та шоурумів із товарами на мапі. У випадку, якщо користувач виявить бажання приїхати і подивитися на товар вживу, ця функція допоможе йому знайти найближчий до нього магазин і побудувати маршрут до нього.

Бувають випадки, коли користувач зацікавлений у певному товарі, проте бажає купити його не у даний момент, а пізніше. Для таких випадків доцільно зробити можливість зберігати вибір товарів, що зараз знаходяться у кошику і дати можливість поновити цей вибір пізніше.

Розглянувши конкурентів і підсумувавши їх переваги та недоліки, можна поставити вимоги до нашого додатку:

- У додатку має бути перелік меблів із їх зображенням та описом
- Має бути можливість переглянути об’ємну модель товару за допомогою віртуальної реальності і маніпулювати нею, за необхідності
- Товари мають мати можливість налаштування своїх певних властивостей (таких як розмір, матеріал, тип ручок й інші) із

автоматичним перерахуванням ціни, відповідно до встановлених значень параметрів.

- Має бути можливість додати товар у кошик, а пізніше, коли кошик сформований можна або оформити замовлення, або ж зберегти його на майбутнє і перейти до наступних покупок
- У наявності має бути мапа із відмітками магазинів та можливістю побудувати маршрут до них

РОЗДІЛ 2 ОГЛЯД ВИКОРИСТАНИХ ТЕХНОЛОГІЙ

Для розробки мобільного застосунку було використано наступні технології:

- Мова програмування Kotlin
- Сервіс Firebase
- Google AR Core
- Бібліотека Jetpack Compose
- Google Maps SDK
- Бібліотека Room
- Бібліотека Hilt

2.1 Мова програмування Kotlin

Для реалізації мобільного застосунку було обрано мову Kotlin. Основними її перевагами є:

1. Сумісність із мовою Java, на якій написано ядро ОС Android і яка була основною мовою для розробки мобільних додатків під Android до 2019р. Це означає, що при написанні коду на Kotlin є можливість напряму викликати код написаний на Java і навпаки.
2. Kotlin має простий синтаксис, подібний до інших мов програмування, таких як Java та C#. Він є простим для вивчення, особливо якщо людина вже мала досвід програмування на інших мовах.
3. Починаючи з 2019р, Google визнала Kotlin пріорітетною мовою розробки під Android. Тобто увесь інструментарій розробки, бібліотеки і подальший розвиток ОС Android будуються таким чином, щоб пришвидшити розробку додатків саме при використанні мови Kotlin.

2.2 Сервіс Firebase

Firebase – це платформа, що розроблена компанією Google для

спрощення і пришвидшення розробки мобільних додатків.

Для збереження даних про товари було використано Firebase Realtime Database. Firebase Realtime Database – це хмарна база даних від Google, що дозоляє зберігати дані на віддаленому сервері у NoSQL базі даних у форматі JSON.

Для збереження фотографій та об’ємних моделей товарів було виеористано Firebase Storage – хмарне сховище, яке дозволяє розробникам додатків завантажувати певні файли та мати до них доступ із мобільного додатку. Це служба зберігання файлів, що дозволяє масштабувати застосунок без втрат у швидкодії.

Також сервіс Firebase мінімізує витрати на збереження даних та файлів, надаючи безоплатний тариф, якого достатньо для невеликих додатків, і, у випадку його перевищення, кошти будуть зніматися тільки за ті ресурси, що були використані, зменшуючи витрати до мінімально можливого значення.

2.3 Google AR Core

Для доповненої реальності було використано Google ARCore[9]. Це набір технологій для використання доповненої реальності у мобільних додатках. Більш того, він доступний не тільки для розробки під Android, а також для iOS, Unity та веб-сайтів.

Google ARCore має інструменти, які дозволяють:

- Відстежувати рухи користувача для коректного співставлення моделі відносно зовнішнього світу
- Розставляти якорі, які відстежують положення об’єкту із часом
- Визначати розмір і розташування поверхонь у навколишньому середовищі
- Розуміти глубину і відстань між поверхнями відносно певної точки

2.4 Бібліотека Jetpack Compose

Для розробки інтерфейсу мобільного додатку було використано технологію Jetpack Compose. Ця технологія прийшла на заміну застарілої технології View, при використанні якої необхідно було створювати інтерфейс додатку в окремих файлах у вигляді XML, а потім використовувати автоматично створені об’єкти у коді.

Jetpack Compose, у свою чергу, пропонує новий, декларативний, підхід до побудови мобільних інтерфейсів. Інтерфейс є функцією від стану. Як тільки зміниться стан, елементи інтерфейсу, що використовували цей стан, також перемалюються. При цьому, якщо елементи не використовували цей стан, перемальовані вони не будуть.

Такий підхід має багато переваг. Перш за все, при створенні інтерфейсу, програміст зобов’язаний продумати, як має виглядати інтерфейс при будь-якому стані програми. Це, на відміну від попереднього імперативного стилю розробки інтерфейсу, дозволяє уникнути ситуацій, коли при певному наборі даних формується зображення, яке не було передбачено при розробці. По-друге, оскільки відмальовкою інтерфейсу займається, по суті, звичайна функція, для розробки можна використовувати будь-які конструкції мови програмування, такі як цикли, умовні вирази, обробку помилок й інші. Це значно спрощує проектування зображення і перевикористання існуючих компонентів.

2.5 Google Maps SDK

Для відображення магазинів і шоурумів на мапі було використано Google Maps SDK[10], що дозволяє інтегрувати функціонал Google Maps до свого додатку.

Вибір саме цього сервісу для відображення об’єктів на мапі обумовлений наступними чинниками:

- На даний момент він є найпопулярнішим сервісом у світі

- Додаток Google Maps наявний на усіх смартфонах, які використовують верифіковану ОС Android, що спрощує інтеграцію із ними.

2.6 Бібліотека Room

Для взаємодії із базою даних було обрано технологію Room. Вона бере на себе піклування про низькорівневі операції над локальною БД SQLite[11-14] і надає програмісту простий декларативний спосіб для оперування над базою даних.

Room є одним із інструментів, що рекомендується компанією Google для розробки мобільних додатків під Android

2.7 Бібліотека Hilt

Для гнучкої, зручної ініціалізації та використання класів, зазвичай використовується такий механізм як ін’єкція залежностей. Для реалізації такого механізму було використано бібліотеку Hilt[15]. Вона дозволяє постачати об’єкти класів у Android-компоненти, враховуючи при цьому життєвий цикл Android-додатку й інші важливі нюанси розробки під Android.

РОЗДІЛ 3 ПРОЕКТУВАННЯ СХЕМИ ЛОКАЛЬНОЇ БАЗИ ДАНИХ

Для підвищення швидкодії додатку і зменшення використання інтернет-трафіку необхідно спроектувати і реалізувати локальну базу даних. Таким чином дані будуть знаходитись на пристрої користувача і швидкість доступу до них буде вищою, аніж у випадку їх отримання з віддаленого серверу.

Для наших вимог була спроектована наступна схема БД, що зображена на рис. 1

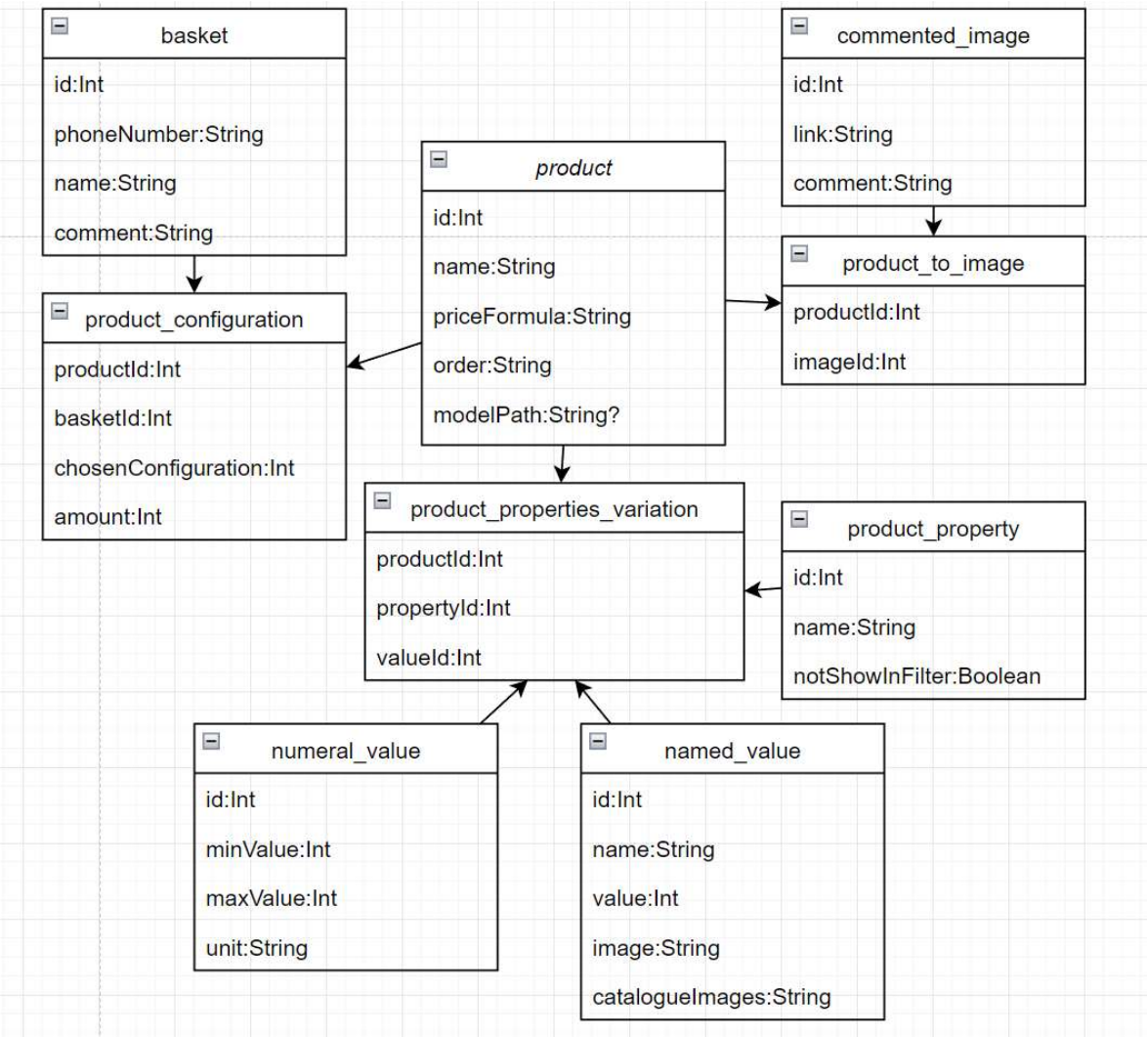

Рисунок 1 – Діаграма БД

У таблицях 1-9 опишемо дані, які будуть зберігатись у таблицях нашої

бази даних

Поле	**Тип**	**Опис**
id	INTEGER	Ключовий атрибут
name	TEXT	Назва товару
priceFormula	TEXT	Формула розрахунки ціни товару в залежності від обраних параметрів товару
order	TEXT	Порядок параметрів товару
modelPath	NULLABLE TEXT	Адреса 3-Д моделі товару

Таблиця 1. Опис таблиці product.

Поле	**Тип**	**Опис**
id	INTEGER	Ключовий атрибут
name	TEXT	Назва параметру
notShowInFilter	BOOLEAN	Чи слід цьому параметру з’являтись у фільтрах

Таблиця 2. Опис таблиці product_property.

Поле	**Тип**	**Опис**
id	INTEGER	Ключовий атрибут

unit	TEXT	Одиниці виміру цього параметру
minValue	INTEGER	Найменше можливе значення цього параметру
maxValue	INTEGER	Найбільше можливе значення цього параметру

Таблиця 3. Опис таблиці numeral_value.

Поле	**Тип**	**Опис**
id	INTEGER	Ключовий атрибут
name	TEXT	Назва цього параметру
value	INTEGER	Значення цього параметру для використання у формулі ціни продукту, якщо оберуть це значення
image	TEXT	Зображення цього параметру для відображення на екрані продукту
catalogueImages	TEXT	Перелік зображень цього параметру для

		відображення на головному екрані

Таблиця 4. Опис таблиці named_value.

Поле	**Тип**	**Опис**
productId	INTEGER	Ключовий атрибут
propertyId	INTEGER	Ключовий атрибут
valueId	INTEGER	Ключовий атрибут

Таблиця 5. Опис таблиці product_properties_variation.

Поле	**Тип**	**Опис**
id	INTEGER	Ключовий атрибут
link	TEXT	Посилання на зображення
comment	TEXT	Коментар до зображення

Таблиця 6. Опис таблиці commented_image.

Поле	**Тип**	**Опис**
productId	INTEGER	Ключовий атрибут
imageId	INTEGER	Ключовий атрибут

Таблиця 7. Опис таблиці product_to_image.

Поле	Тип	Опис
id	INTEGER	Ключовий атрибут
phoneNumber	TEXT	Номер телефону, який буде використовуватись при оформленні замовлення із цього кошика
name	TEXT	Ім'я, яке буде використовуватись при оформленні замовлення із цього кошика
comment	TEXT	Коментар до замовлення

Таблиця 8. Опис таблиці basket.

Поле	Тип	Опис
productId	INTEGER	Ключовий атрибут
basketId	INTEGER	Ключовий атрибут
chosenConfiguration	TEXT	Закодована конфігурація товару
amount	INTEGER	Кількість обраного товару у корзині

Таблиця 9. Опис таблиці product_configuration.

РОЗДІЛ 4 РОЗРОБКА МОБІЛЬНОГО ДОДАТКУ

При розробці було використано архітектурний шаблон проектування під назвою MVVM. Цей шаблон дозволяє розділити код, що відповідає за створення інтерфейсу від власне логіки додатку і від логіки отримання і збереження даних.

Таким чином, можна виділити декілька типів класів у наші програмі. Ті, що відповідають за створення інтерфейсу, зазвичай мають наступну назву ***Fragment або ж ***Activity. Ті, що відповідають за логіку поведінки додатку зазвичай мають таку назву ***ViewModel. Ті ж, що відповідають за отримання і збереження даних використовуються класами з назвою ***Repository.

4.1 Створення класів для представлення даних для використання по всьому додатку

Для зручного оперування даними та їх використання були створені наступні класи: *Product, ProductProperty, PropertyValue, ProductPropertiesSet, NamedValue, NumeralValue, MultiLanguageString, CommentedImage, Basket.*

Розглянемо клас *Product*. У ньому знаходяться поля, що необхідні для коректного відображення продукту і взіємодії з ним.

```
data class Product(
    var id: Int,
    var name: MultiLanguageString,
    var priceFormula: String,
    val imageUrl: List<CommentedImage>,
    var propertiesList: Map<ProductProperty, List<PropertyValue>>,
    var order: List<Int>,
    var chosenConfiguration: ProductPropertiesSet? = null,
    var amount: Int = 1,
    var modelPath: String? = null
) {
    fun configurationToPresentableString(): String {
```

```
        var result: String = ""
        var isFirst = true
        val properties = propertiesList.keys
        for (i in chosenConfiguration?.set ?: mapOf()) {
            if (!isFirst) {
                result += '\n'
            }
            result += "${properties.find { it.id == i.key }?.name?.getText() ?: ""} : ${i.value.getPropertyName()}"
            isFirst = false
        }
        return result
    }
}
```

Аналогічним чином виглядають й інші класи моделей.

4.2 Отримання даних із віддаленого серверу

Для роботи застосунку необхідно отримувати дані, що знаходяться на віддаленому сервері. Наведемо приклад отримання інформації про продукти:

```
override suspend fun getAllProducts(): List<ProductDTO> {
    val url = HttpUrl.Builder().scheme("https")
        .host(SERVER_BASE_URL)
        .addPathSegment("products$JSON")
        .build()
    val request = Request.Builder().url(url).build()
    val call = mClient.newCall(request)
    try {
        val response = call.execute()
        if (response.isSuccessful) {
            val json = response.body?.string()
            return if (json != null) {
                getProductsListFromJSON(json)
            } else {
                listOf()
            }
        }
    } catch (e: Exception) {
        return listOf()
    }
```

```
    return listOf()
}

private fun getProductsListFromJSON(string: String): List<ProductDTO> {
    val json = JSONArray(string)
    val result = mutableListOf<ProductDTO>()
    for (i in 0 until json.length()) {
        val value = json[i]
        if (value is JSONObject) {
            val gson = GsonBuilder().create()
            val product = gson.fromJson(value.toString(), ProductNetworkDto::class.java)
            result.add(product.toProductDto())
        }
    }
    return result
}
```

Для створення належного рівня незалежності внутрішніх моделей додатку від моделей, що отримуються від серверу, було створено класи для репрезентації моделей, що прийшли від віддаленого серверу і подальшого їх перетворення у класи, що використовуються усередині додатку.

Приклад такого класу *ProductNetworkDto*, що у подальшому перетворюється у клас *Product*:

```
class ProductNetworkDto(
    val id: Int,
    var name: Map<String, String>,
    @SerializedName("price_formula")
    var priceFormula: String,
    @SerializedName("images")
    val imageUrl: List<Int>,
    @SerializedName("properties_list")
    var propertiesList: List<PropertyToVariationsNetworkDto>,
    @SerializedName("androidModelPath")
    var modelPath: String? = null
) {
    fun toProductDto(): ProductDTO {
        val map = mutableMapOf<Int, List<Int>>()
        val order = mutableListOf<Int>()
```

```
        for (i in propertiesList) {
            map[i.id] = i.variations
            order.add(i.id)
        }
        return ProductDTO(
            id = id,
            name = MultiLanguageString(name),
            priceFormula = priceFormula,
            order = order,
            imageUrl = imageUrl,
            propertiesList = map,
            modelPath = modelPath
        )
    }
}
```

Інші моделі для даних, що прийшли з серверу виглядають аналогічно

Для зменшення використання інтернет-трафіку клієнта, збільшення швидкодії додатку та ефективного кешування даних, на віддаленому сервері також знаходиться дата останньої модифікації даних. Для її отримання використовується наступна функція:

```
override fun getTimestamp(): Long {
    val url = HttpUrl.Builder().scheme("https")
        .host(SERVER_BASE_URL)
        .addPathSegment("timestamp$JSON")
        .build()
    val request = Request.Builder().url(url).build()
    val call = mClient.newCall(request)
    try {
        val response = call.execute()
        if (response.isSuccessful) {
            val json = response.body?.string()
            return json?.toLong() ?: 0
        } else {
            throw InternetConnectionErrorException()
        }
    } catch (e: InternetConnectionErrorException) {
        throw e
    } catch (e: Exception) {
```

```
        throw InternetConnectionErrorException()
    }
}
```

У подальшому, дата, отримана після завершення цього запиту порівнюється із датою останнього разу завантажання даних. Якщо дата оновлення даних є новішою, закешовані дані видаляються і нові дані завантажуються наново. Оскільки каталог меблів буде змінюватись значно рідше, аніж передивлятись, такий підхід дозволяє значно зменшити навантаження на віддалений сервер і, відповідно, зекономити значну кількість грошей.

4.3 Взаємодія із БД

Створення БД із використанням бібліотеки Room складається із наступних кроків:

- Створення моделей для збереження
- Визначення операцій, які будуть здійнюватись над БД
- Створення об’єкту бази даних, який буде використовуватись для взаємодії з БД

4.3.1 Створення моделей для збереження у БД

Для збереження даних у БД необхідно певним чином створити класи моделей.

Наведемо приклад класу, що буде відповідати таблиці product у схемі БД, що була наведена у розділі 3:

```
@Entity(tableName = "product")
data class ProductDTO(
    @PrimaryKey
    val id: Int,
    var name: MultiLanguageString,
    var priceFormula: String,
    var order: List<Int>,
```

```
    var modelPath: String? = null,
    @Ignore
    var propertiesList: Map<Int, List<Int>>
) {
    constructor(
        id: Int,
        name: MultiLanguageString,
        priceFormula: String,
        order: List<Int>,
        modelPath: String?,
        imageUrl: List<Int>,
    ) : this(id, name, priceFormula, order, modelPath, imageUrl, mapOf())
}
```

Для вказання певних параметрів таблиці для БД, використовуються анотації.

Анотація @Entity(tableName = "product") вказує, що цей клас є класом-моделлю, яка буде зберігатись у БД, причому ім'я аблиці для збереження має бути product.

Анотація @PrimaryKey вказує, що дане поле є ключовим у таблиці.

Анотація @Ignore вказує, що дане поле має бути проігнороване при збереженні у БД і не має ніяким чином бути відображене у таблиці. Це поле використовується в подальшому для більш зручного перетворення даних із цієї моделі у основну модель Product.

Також, оскільки у цьому класі є поле із анотацією @Ignore нам необхідно дати можливість створити цей клас без цього поля. Саме для цього було створенно вторинний конструктор, який не потребує цього поля.

Інші класи-моделі, що репрезентують таблиці у схемі БД для додатку, були створені аналогічним чином.

4.3.2 Визначення операцій, які будуть здійнюватись над БД

Для проведення операцій над локальною БД необхідно створити інтерфейс взаємодії. Нижче наведено приклад коду такого інтерфейсу для

операцій над моделлю ProductDTO:

```
@Dao
interface ProductDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insert(item: ProductDTO)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insert(items: List<ProductDTO>)

    @Query("select * from product")
    suspend fun getAll(): List<ProductDTO>

    @Query("select * from product where id in (:ids)")
    suspend fun getByIds(ids: List<Int>): List<ProductDTO>

    @Query("select * from product where id ==:id")
    fun getLiveById(id: Int): LiveData<ProductDTO?>

    @Query("delete from product")
    fun deleteAll()

    @Delete
    fun delete(item: ProductDTO)

    @Query("select * from product where id in (select productId from productpropertiesvariation where valueId in
(:ids))")
    suspend fun getProductsFromPropertiesValueIds(ids: List<Int>): List<ProductDTO>

    @Transaction
    suspend fun getByIdsBulk(ids: List<Int>): List<ProductDTO> {
        return performBulkActionSuspend(ids, ::getByIds)
    }

    @Transaction
    suspend fun getProductsFromPropertiesValueIdsBulk(ids: List<Int>): List<ProductDTO> {
        return performBulkActionSuspend(ids, ::getProductsFromPropertiesValueIds)
    }
}
```

Можна побачити, що він являє собою інтерфейс із декларацією сигнатур

функцій і певними анотаціями. В залежності від сигнатури функції та анотацій, Room проводить відповідні дії над локальною БД і повертає відповідний результат.

Анотація @Dao означає, що цей інтерфейс є інтерфейсом взаємодії із БД.

Анотація @Insert(onConflict = OnConflictStrategy.*REPLACE*) означає, що дана функція виконує вставку даних у БД, а параметр OnConflictStrategy.*REPLACE* означає, що у випадку, коли дані із таким ключовим значенням уже присутні у таблиці, їх треба оновити новими значеннями.

Анотація @Delete означає, що дана функція видаляє дані із таблиці.

Анотація @Query("…") означає, що дана функція робить певний запит до БД із використанням запиту SQL, що написаний у параметрах анотації. Наприклад, функція

```
@Query("select * from product where id in (:ids)")
 suspend fun getByIds(ids: List<Int>): List<ProductDTO>
```

обирає усі продукти, ключовий атрибут яких має одне із значень, що були передані як параметр при виклику цієї функції.

Анотація @Transaction означає, що дана функція виконує транзакцію над БД і якщо який-небудь крок транзакції буде невдалим, усі зміни, що були зроблені раніше усередині цієї транзакції, будуть скасовані і БД буде повернута до стану, що був до початку цієї транзакції.

Інтерфейси для взаємодії з іншими таблицями БД побудовані аналогічним чином.

4.3.3 Створення об’єкту бази даних, який буде використовуватись для взаємодії з БД

Клас, що використовується для взаємодії з БД, виглядає наступним чином:

```
@Database(
    entities = [
```

```
        BasketDTO::class,
        ProductDTO::class,
        ProductPropertyDTO::class,
        NamedValue::class,
        NumeralValue::class,
        ProductPropertiesVariation::class,
        ProductToImage::class,
        CommentedImage::class,
        ProductConfiguration::class,
    ],
    version = 2
)
@TypeConverters(Converters::class)
abstract class AppDB : RoomDatabase() {
    companion object {
        const val DATABASE_NAME = "app_db"
        const val MAX_LIST_SIZE = 900
    }

    abstract fun getBasketDao(): BasketDao
    abstract fun getProductToImageDao(): ProductToImageDao
    abstract fun getCommentedImageDao(): CommentedImageDao
    abstract fun getProductDao(): ProductDao
    abstract fun getPropertyDao(): PropertyDao
    abstract fun getTypedValueDao(): NamedValueDao
    abstract fun getNumeralValueDao(): NumeralValueDao
    abstract fun getProductToPropertyDao(): ProductPropertiesVariationDao
    abstract fun getProductConfigurationDao(): ProductConfigurationDao
}
```

Тут анотація @Database означає, що цей клас репрезентує базу даних. У параметри цієї анотації передається перелік класів-моделей, для яких треба створити таблиці у локальній БД.

Абстрактні методи get***Dao() створені для отримання інтерфейсів взаємодії з БД, що були описані у попередньому пункті.

Об’єкт цього класу створюється наступним чином:

```
Room.databaseBuilder(
    context,
    AppDB::class.java, AppDB.DATABASE_NAME
```

```
)
    .fallbackToDestructiveMigration()
    .build()
```

Потрібно зауважити, що цей об’єкт має бути єдиним, для запобігання конфліктів при зміні даних у БД із різних місць.

4.4 Побудова основної структури додатку

Для ін’єкції залежностей у класи додатку за допомогою бібліотеки Hilt, потрібно додати анотацію @HiltAndroidApp до класу Application. Після цього потрібно створити модулі, у яких буде відбуватися створення необхідних об'єктів. Приклад модуля для створення об'єкту БД наведено нижче:

```
@Module
@InstallIn(SingletonComponent::class)
object RoomModule {

    @Singleton
    @Provides
    fun provideDB(@ApplicationContext context: Context): AppDB {
        return Room.databaseBuilder(
            context,
            AppDB::class.java, AppDB.DATABASE_NAME
        )
            .fallbackToDestructiveMigration()
            .build()
    }
}
```

Функція, що постачає об’єкт БД помічена анотацією @Singleton, що означає, що цей об’єкт буде створено один раз і при постачанні буде використаний цей єдиний екземпляр класу.

Надалі, при необхідності отримання об'єкту певного класу потрібно помітити анотацією @Inject поле чи конструктор, у який потрібно передати об’єкт і він буде переданий автоматично. Додатково, якщо потрібно передати об’єкт у клас ViewModel його треба помітити анотацією @HiltViewModel, а якщо

у клас Fragment чи Activity – @AndroidEntryPoint

Таким чином, створення усіх об’єктів відбувається в окремому місці і, при потребі замінити один об’єкт іншим, це буде зроблено в одному місці, без зміни інших частин коду.

Згідно до обраної архітектури, взаємодія з даними повинна бути абстрагована за допомогою класів з назвою ***Reposiory. У якості прикладу такого класу покажемо інтерфейс ProductsRepository:

```
interface ProductsRepository {
    val isDataUpdated: LiveData<Boolean>
    val errorStates: LiveData<ErrorStates>
    val placeSameOrderState: LiveData<DataLoadState<Boolean>>
    fun resetPlaceSameOrderState()
    fun getLiveProductByName(productId: Int): LiveData<ProductDTO?>
    suspend fun getProductsByIds(products: List<Int>): List<Product>
    suspend fun getProductsByValueIds(ids: List<Int>): List<Product>
    suspend fun productDtoToProduct(products: List<ProductDTO>): List<Product>
    suspend fun portfolioDtoToPortfolio(products: List<PortfolioDTO>): List<Portfolio>
    suspend fun orderSamePortfolio(name: String, email: String, phone: String, portfolioId: Int)
    fun getAllProducts(): LiveData<List<Product>>
    fun getAllPortfolios(): LiveData<List<PortfolioDTO>>
    fun getCollection(id: Int): LiveData<NamedValue>
}

sealed class ErrorStates() {
    object AllRight : ErrorStates()
    object VersionIncorrect : ErrorStates()
    object InternetConnectionError : ErrorStates()
}
```

Клас, що реалізує цей інтерфейс бере дані з локальної БД, а якщо вони відсутні, завантажує їх з віддаленого серверу, зберігає до локальної БД і повертає у відповідному форматі.

У подальшому класи такого типу використовуються у класах, що відповідають за основну логіку додатку. Вони мають назву ***ViewModel. Приклад такого класу наведено нижче

```
@HiltViewModel
class CatalogueViewModel @Inject constructor(private val mProductsRepository: ProductsRepository) :
    ViewModel() {
    private val mProductsLiveData: LiveData<List<Product>> = mProductsRepository.getAllProducts()
    private var mAllProducts = listOf<Product>()
    val allProperties = mutableStateOf(mapOf<ProductProperty, List<PropertyValue>>())
    val chosenPropertiesValuesIds = mutableStateOf(mutableListOf<Int>())
    val productsToPresent = mutableStateOf(listOf<Product>())
    val collections = mutableStateOf(listOf<PropertyValue>())

    init {
        mProductsLiveData.observeForever {
            mAllProducts = it
            allProperties.value = getProperties(it)
            collections.value = getCollections(it)
            updateProducts()
        }
    }

    fun clearFilters() {
        chosenPropertiesValuesIds.value = mutableListOf()
    }

    fun updateProducts() {
        viewModelScope.launch {
            if (chosenPropertiesValuesIds.value.isEmpty()) {
                productsToPresent.value = mAllProducts
            } else {
                withContext(IO) {
                    val products =
                        mProductsRepository.getProductsByValueIds(chosenPropertiesValuesIds.value)
                    productsToPresent.value = products
                }
            }
        }
    }

    fun addChosenValue(id: Int) {
        val result = mutableListOf<Int>()
        result.addAll(chosenPropertiesValuesIds.value)
        result.add(id)
```

```
        chosenPropertiesValuesIds.value = result
        updateProducts()
    }

    fun removeChosenValue(id: Int) {
        val result = mutableListOf<Int>()
        result.addAll(chosenPropertiesValuesIds.value)
        result.remove(id)
        chosenPropertiesValuesIds.value = result
        updateProducts()
    }

    private fun getCollections(list: List<Product>): List<PropertyValue> {
        val result = mutableSetOf<PropertyValue>()
        for (i in list) {
            try {
                result.addAll(getValuesByPropertyId(i.propertiesList, 1))
            } catch (e: WrongDataException) {
                continue
            }
        }
        return result.toList()
    }

    private fun getProperties(products: List<Product>): Map<ProductProperty, List<PropertyValue>> {
        val result = mutableMapOf<ProductProperty, MutableList<PropertyValue>>()
        for (i in products) {
            for (t in i.propertiesList)
                if (result[t.key] == null) {
                    result[t.key] = t.value.toMutableList()
                } else {
                    result[t.key]!!.addAll(t.value.minus(result[t.key]!!))
                }
        }
        return result
    }
}
```

Можна побачити, що цей клас бере дані з відповідного репозиторію, після чого певним чином змінює їх для коректного їх відображення. Завдяки технології Jetpack Compose, усі зміни у даних, що були створені за допомогою

функції *mutableStateOf*() будуть миттєво відображені у інтерфейсі користувача.

Окрім того, у Android, клас ViewModel використовується для збереження стану програми у випадках, коли її інтерфейс вимушений бути знищений. Наприклад, якщо користувач поверне телефон на 90 градусів, інтерфейс буде створений наново, що може призвести до втрати певних даних. Клас ViewModel розроблений саме для того, щоб у таких і подібних випадках дані були збережені.

4.5 Створення інтерфейсу застосунку

Застосунок має два активіті[16]: MainActivity та ArActivity

ArActivity являє собою екран із доповненою реальністю.

MainActivity містить у собі майже усі екрани, якими буде користуватись користувач. Ці екрани будуть реалізовані у вигляді фрагментів[16], які будуть змінювати один одного в залежності від дій користувача.

Для навігацією між екранами було використана бібліотека Jetpack Navigation Component[17], що була створена компанією Google і є рекомендованим рішенням для навігації для ОС Android. Вона дозволяє побудувати граф навігації між екранами і в подальшому переходити між ними за необхідності. Також при переході між екранами можна передавати певні дані, які допоможуть відкрити новий екран коректно. Наприклад, при відкритті ерану конкретного товару, можна передати його id і тоді достатньо буде взяти товар із конкретним id і відобразити його.

Екрани, що були створені:

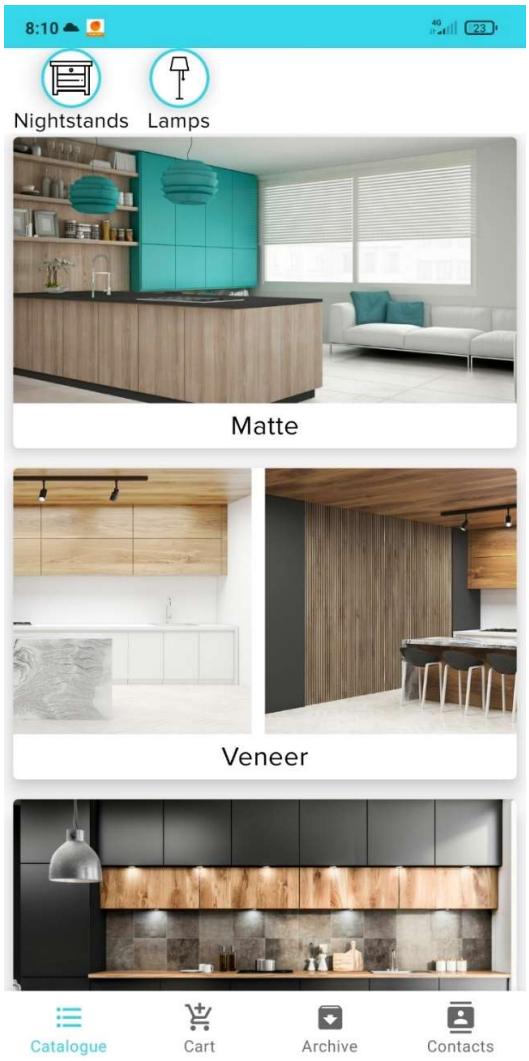

Рисунок 2 – Екран каталогу

На рис. 2 зображено перший екран, який бачить користувач при відкритті нашого мобільного додатку. Згори розташований перелік типів товарів, які доступні в цьому магазині. При натисканні на певний тип відкривається екран з переліком товарів цього типу

Нижче розташовані колекції, які доступні у цьому магазині, для того, щоб клієнт міг побачити, який результат він може отримати, якщо замовити декілька товарів з однієї колекції. При необхідності, користувач може натиснути на зображення колекції. В такому випадку воно відкриється на покний екран і користувач зможе масштабувати його, за необхідності.

На прикладі цього екрану покажемо, як будується інтерфейс за допомогою технології Jetpack Compose.

Розглянемо цей екран (без нижнього меню, оскілки воно є спільним елементом усіх екранів). Його структуру можна описати наступним чином:

Згори у нього знаходиться список круглих зображень, які йдуть один за одним горизонтально. Нижче знаходиться вертикальний список елементів, усередині кожного з яких є горизонтальний список елементів.

Для поставлення елементів один під одним використовується функція Column().

Для відображення списку елементів вертикально чи горизонтально використовуються функції LazyColumn() та LazyRow() відповідно.

Тепер, використавши ці знання, можна творити необхідний нам екран (частина коду була опущена для демонстраційних цілей):

```
@Composable
private fun MainContent() {
   Column() {
            val typedProperties = viewModel.allProperties.value
      LazyRow() {
         items(typedProperties.size) { i ->
            val item = typedProperties[i]
            ItemTypeLayout(item = item)
         }
      }
      LazyColumn(modifier = Modifier.fillMaxHeight(1f)) {
         itemsIndexed(items = collectionValues) { index, collection ->
            CollectionLayout(collection)
         }
      }
   }
}

@Composable
private fun ItemTypeLayout(item: NamedValue, modifier: Modifier, imageWidth: Dp) {
   Column(modifier = modifier) {
```

```
        Card(
            shape = CircleShape,
            border = BorderStroke(2.dp, MaterialTheme.colors.primary),
        ) {
            val image = ImageUtils.loadPicture(
                url = item.catalogueImages[0],
                defaultImage = R.drawable.ic_launcher_background,
            ).value!!
            Image(bitmap = image.asImageBitmap())
        }
        Text(
            text = item.getPropertyName(),
            color = MaterialTheme.colors.onBackground,
            fontSize = 16.sp,
        )
    }
}

@Composable
private fun ItemLocationLayout(item: NamedValue, modifier: Modifier, imageWidth: Dp) {
    if (item.catalogueImages.isNotEmpty()) {
        Column(modifier = modifier) {
            Card(border = BorderStroke(2.dp, MaterialTheme.colors.primary)) {
                val image = ImageUtils.loadPicture(
                    url = item.catalogueImages[0],
                    defaultImage = R.drawable.ic_launcher_background,
                ).value
                image?.let { Image(bitmap = it.asImageBitmap()) }
            }
            Text(
                text = item.getPropertyName(),
                color = MaterialTheme.colors.onBackground,
                fontSize = 16.sp,
            )
        }
    }
}

@Composable
private fun CollectionLayout(collection: NamedValue) {
    Column {
```

```
        Card {
            Column {
                LazyRow {
                    itemsIndexed(collection.catalogueImages) { position, item ->
                        val image = ImageUtils.loadPicture(
                            url = item,
                            imageHeight = LocalDensity.current.run { height.toPx() }
                                .roundToInt(),
                            defaultImage = R.drawable.ic_launcher_background,
                        ).value!!
                        Image(bitmap = image.asImageBitmap())
                    }
                }
                Text(
                    text = collection.name.getText(),
                    fontSize = 20.sp,
                    color = MaterialTheme.colors.onSurface,
                )
            }
        }
        Spacer(modifier = Modifier.height(7.dp))
    }
}
```

Анотація @Composable над функціями вказує, що ця функція є функцією, що відмальовує певний елемент інтерфейсу.

Також важливим елементом для побудови інтерфейсу є параметер modifier. Цей параметер відповідає за розміри, місцеположення, фоновий колір елементу, за взаємодую із цим елементом із використанням натискань, жестів й іншими способами взаємодії, а також за багато інших функцій, які можуть вплинути на поведінку конкретного елементу на екрані смартфону.

Інтерфейс інших екранів побудований аналогічним чином.

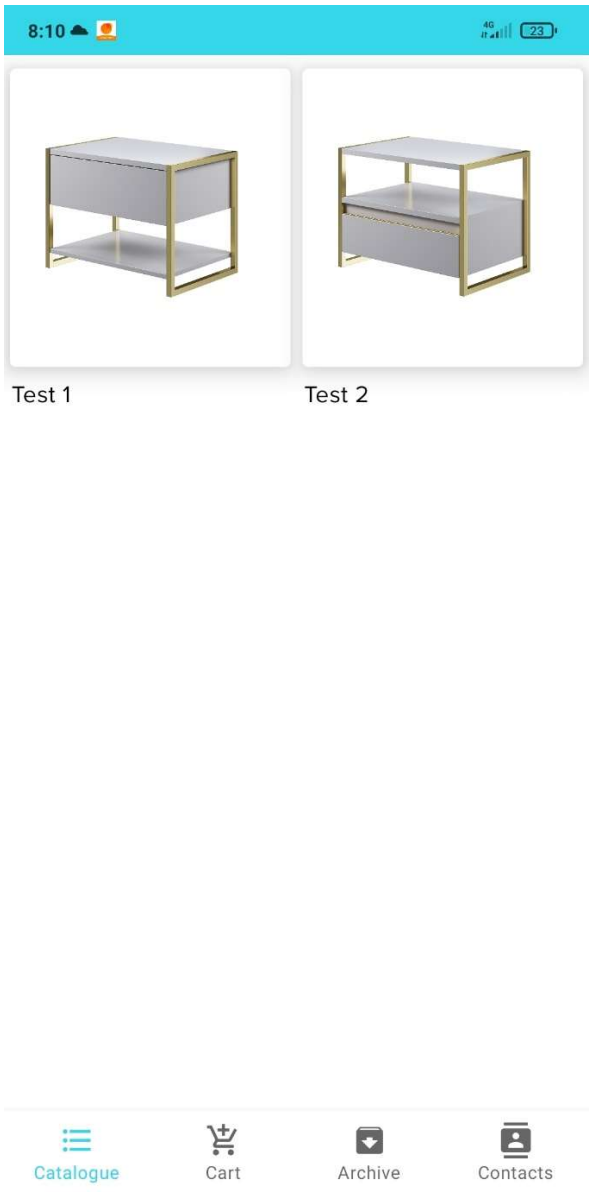

Рисунок 3 – Екран переліку товарів певного типу

На рис. 3 зображено екран переліку товарів певного типу, що буде відкритий, якщо на екрані каталогу користувач обере певний тип товару. На цьому екрані знаходяться усі товари певного типу. На картці товару можна побачити його назву та зображення. При натисканні на один із товарів відкривається екран цього конкретного товару.

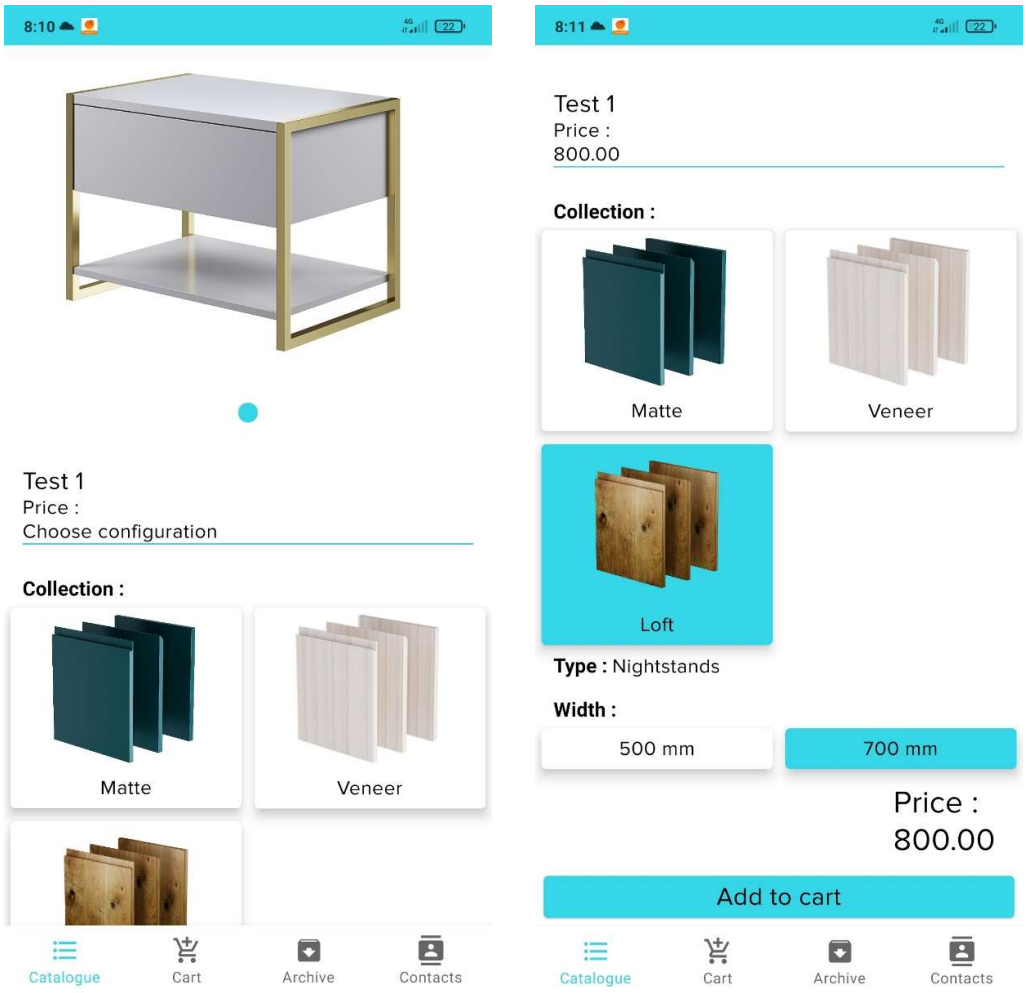

Рисунок 4 – Екран конкретного товару

На рис. 4 зображено екран конкретного товару. Цей екран показується коли людина обрала певний товар для того, щоб роздивитись його детальніше.

Згори знаходяться зображення товару. Їх можна свайпати щоб роздивитися усі.

Нижче знаходяться параметри товару серед яких можна обрати. Параметри можуть бути якісними чи кількісними. Як видно з рисунків 4 та 5, якісні значення можуть мати певне зображення для кращого розуміння, що саме мається на увазі під цим значенням. При виборі необхідної конфігурації товару, автоматично рахується ціна по заданій формулі.

Нижче знаходиться розрахована ціна товару і кнопка, що додає товар до

кошику.

Однією із вимог до нашої програми є автоматичне перерахування ціни товару в залежності від обраних значень його параметрів. Для коректного розпізнавання формули ціни товару а також подальшого підрахунку ціни використовується технологія ANTLR[18].

Рисунок 5 – Екран доповненої реальності

На рис. 5 зображено екран доповненої реальності. На цей екран можна потрапити, натиснувши відповідну кнопку на екрані товару.

На цьому екрані користувач може поставити обраний товар у те місце, де він бажає його бачити, перемістити, покрутити, змінити розмір моделі товару, а також походити і роздивитись обраний товар з усіх сторін.

Для інтеграції технології доповненої реальності було використано технологію Google AR Core. Для її роботи, потрібно мати файл, що являє

собою трьохвимірну модель об’єкту. Файли можуть бути формату .obj, .fbx, .glb й інших. Для використання нашим додатком було використано формат .glb, оскільки він є більш новим і створювався із метою бути більш гнучким і оптимізовани, ніх старі формати. Він має менший розмір і дозволяє у майбутньому простіше додати текстури до моделей.

Код частини логіки використання доповненої реальності наведено нижче:

```
private fun startAr() {
    if (!hasCameraPermission()) {
        requestCameraPermission()
        return
    }
    val values = navArgs<ArActivityArgs>()
    val file = File.createTempFile("000" + values.value.id.toString(), ".glb")
    ServerRequestsImpl().downloadFile(values.value.link, file) {
        buildModel(file)

        mArFragment.setOnTapArPlaneListener { hitResult, plane, motionEvent ->
            if (mObject == null) {
                val ancorNode = AnchorNode(hitResult.createAnchor())
                val obj = TransformableNode(mArFragment.transformationSystem)
                obj.renderable = mModelRenderable
                obj.scaleController.minScale = 0.1f
                obj.scaleController.maxScale = 1f
                obj.rotationController
                obj.setParent(ancorNode)
                mArFragment.arSceneView.scene.addChild(ancorNode)
                mObject = obj
                obj.select()
            } else {
                mObject!!.select()
            }
        }
    }
}
fun buildModel(file: File) {
    val renderableSource = RenderableSource.builder()
```

```
        .setSource(this, Uri.parse(file.path), RenderableSource.SourceType.GLB)
        .setRecenterMode(RenderableSource.RecenterMode.ROOT)
        .build()
    ModelRenderable.builder().setSource(this, renderableSource).setRegistryId(file.path).build()
        .thenAccept { modelRenderable ->
          mModelRenderable = modelRenderable
        }
}
```

Загальна логіка дій цього коду наступна: З віддаленого серверу завантажується файл формату .glb, який у подальшому використовується для створення об’єкту трьохвимірної моделі у доповненій реальності. Коли користувач натискає на підлогу, у місці, куди він натиснув, створюється трьохвимірна модель обраного товару, якою користувач може оперувати, здійснюючи її переміщення, поворот та зміну її розмірів за допомогою відповідних жестів, здійснених на екрані його мобільного телефону.

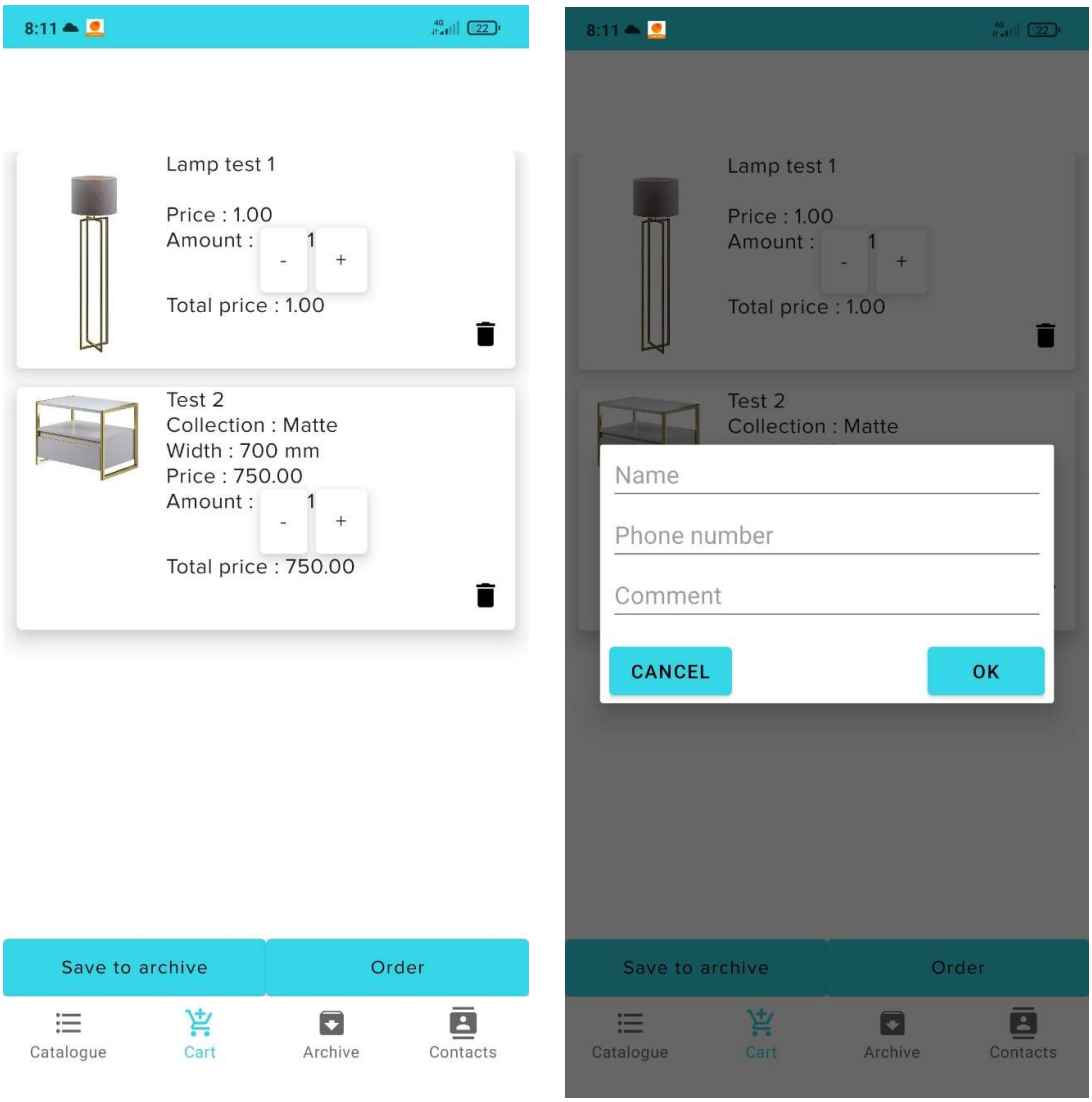

Рисунок 6 – Екран кошику: а – стан редагування; б – стан замовлення

На рис. 6 знаходиться екран кошику. Потрапити на цей екран можна натиснувши на відповідну кнопку у нижньому меню додатку, або ж обравши певний кошик з архіву.

На цьому екрані знаходяться товари, що були додані до цього кошику. На картці товару видно зображення, обрана конфігурація, кількість та загальна ціна цього товару. Тут можна редагувати їх кількість, прибирати їх, або ж натиснути на конкретний товар і змінити його конфігурацію. Однакові товари з різною конфігурацією відображаються окремо, щоб не було непорозумінь.

Нижче знаходяться кнопки для додавання корзини до архіву та для

оформлення замовлення. При натисканні, обидві показують діалог, у який можна ввести ім’я, номер телефону та коментар. Після підтвердження, корзина або зберігається до архіву, а актуальною корзиною стає нова, якщо було натиснуто на кнопку додавання корзини до архіву, або ж, якщо була натиснута кнопка для оформлення замовлення, замовлення відправляється до менеджера, який в подальшому контактує з користувачем.

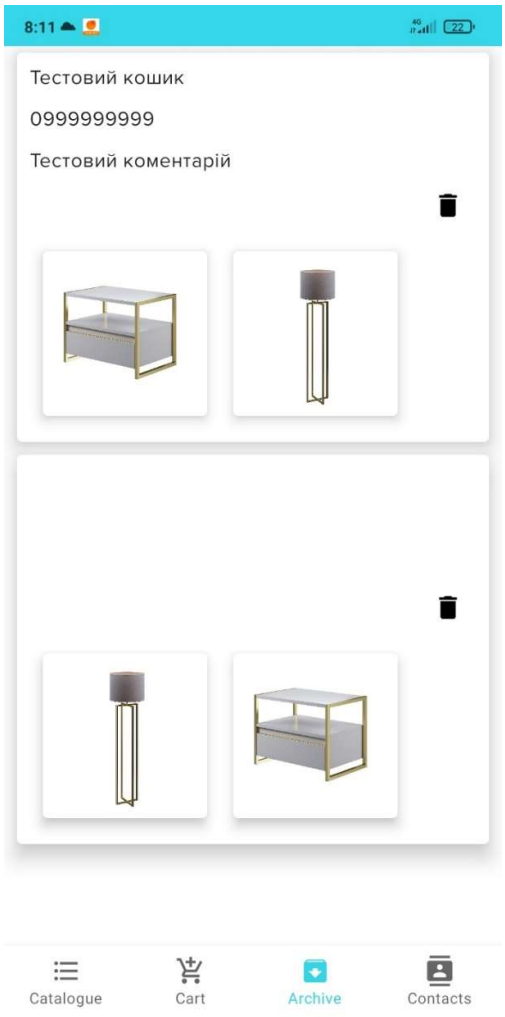

Рисунок 7 – Екран архіву

Екран архіву зображений на рис. 7. На цей екран можна потрапити, натиснувши відповідну кнопку у нижньому меню додатку.

На цьому екрані можна подивитися усі кошики, для яких не було оформлено замовлення і, при необхідності, відредагувати певний кошик, або зробити актуальним. Актуальний коршик означає, що саме до нього будуть

додаватися нові товари, які обере користувач.

На кожній картці корзини видно перелік товарів, що у ній знаходяться, а також її назва, коментар і контактні дані замовника.

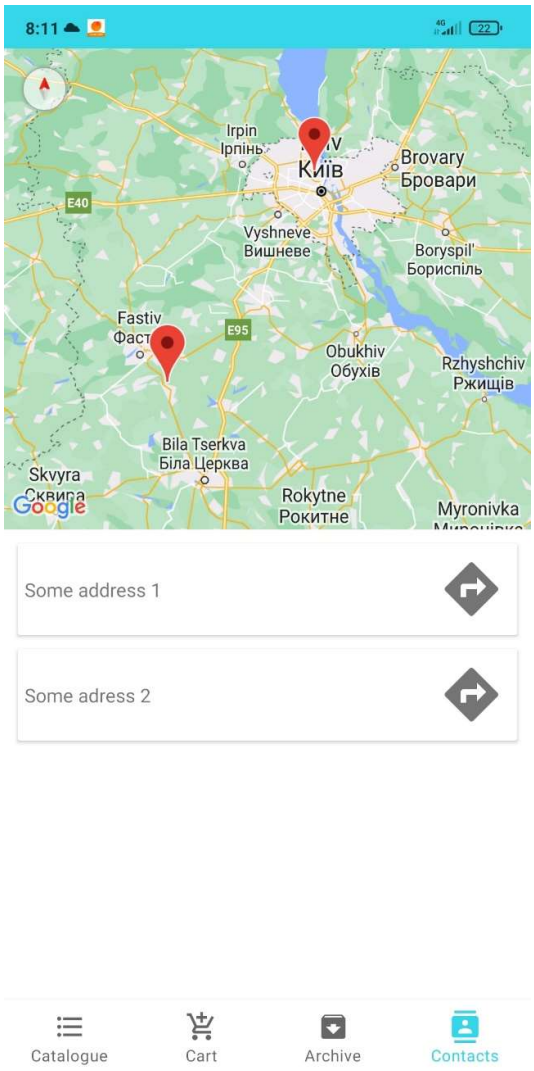

Рисунок 8 – Екран контактів

На рис. 8 можна побачити екран контактів. На цей екран можна потрапити, натиснувши відповідну кнопку у нижньому меню додатку.

На цьому екрані знаходиться мапа з місцеположенням магазинів, їх описом та контактами. При натисканні на конкретний магазин він буде відображатися на карті, а при натисканні на знак справа біля магазину, користувач буде направлений у Google Maps і там буде побудований маршрут до цього магазину.

Для розробки цього екрану було використано Google Maps SDK.

Для побудови маршрутів було використано Google Maps API. Код, що допомогає побудувати маршрут виглядає наступним чином:

```
val gmmIntentUri =
    Uri.parse("google.navigation:q=${marker.latitude},${marker.longitude}")
val mapIntent = Intent(Intent.ACTION_VIEW, gmmIntentUri)
mapIntent.setPackage("com.google.android.apps.maps")
if (mapIntent.resolveActivity(requireActivity().packageManager) != null) {
    startActivity(mapIntent)
}
```

Тут будується URI з координатами точки на мапі, після чого створюється намір відкрити цю URI і віддається ОС Android для його задоволення. Потім цей намір отримує Google Maps і будує маршрут, що у ньому вказаний.

ВИСНОВКИ

У процесі виконання даної роботи досліджено існуючі додатки магазинів з продажу меблів, внаслідок чого сформовано вимоги до мобільного додатку.

Для розробки мобільного додатку, обрано перелік інструментів та технологій, що були використані для розробки додатку, а також спроектована схема локальної бази даних, що створюється і використовується на пристрої користувача.

Розроблений мобільний додаток, у якому використовується технологія доповненої реальності, що дозволяє користувачам додатку подивитися товар, що їм сподобався, у інтер'єрі своєї кімнати і на підставі побаченого зробити рішення, чи купувати їм цей товар. Інтегровано технологію Google Maps, що дозволяє користувачам переглянути локації магазинів з товарами і, за бажанням, побудувати маршрут до обраного магазину. У результаті отримано робочий додаток, що задовольняє початковим вимогам.

Підводячи підсумки роботи, зауважимо, що досягнено завдань, поставлених на початку роботи. Використано актуальні технології розробки мобільних додатків для ОС Android та створений працюючий застосунок з потрібною функціональністю. Отримано досвід розробки мобільних додатків під ОС Android та інтеграції технології доповненої реальності.

Продовжити подальшу розробку додатку можна у декількох напрямках:

- Додати можливість використовувати трьохвимірні моделі з текстурами
- Реалізація можливості редагування моделі об'єкту відповідно до наявного інтер'єру кімнати і подальше замовлення товару, що відповідає відредагованій моделі
- Реалізація можливості розміщувати одразу декілька різних товарів у своїй кімнаті, що буде корисним, якщо користувач планує замовити одразу великий перелік меблів у свою кімнату

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. ОС Android [Електронний ресурс] – Режим доступу до ресурсу: https://www.android.com/.
2. Мова програмування Kotlin [Електронний ресурс] – Режим доступу до ресурсу: https://kotlinlang.org/.
3. Jemerov D., Isakova S. Kotlin in action. Manning Publications Co. LLC, 2017. 360 с.
4. Firebase Realtime Database [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/database.
5. Firebase Storage [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs/storage
6. Room [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/jetpack/androidx/releases/room.
7. Jepack Compose [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/jetpack/compose.
8. Castillo J. Jetpack Compose internals / Jorge Castillo., 2022.
9. Google AR Core [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/ar.
10. Google Maps SDK [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/maps/documentation/android-sdk/overview.
11. SQLite [Електронний ресурс] – Режим доступу до ресурсу: https://sqlite.org/index.html.
12. Aditya S. K., Karn V. K. Android SQLite Essentials. Packt Publishing, 2014. 110 с.
13. Sribatsa D. SQLite for mobile apps simplified. CreateSpace Independent Publishing Platform, 2014. 122 с.
14. Kreibich J. A. Using SQLite. O'Reilly Media, Incorporated, 2010, 528 с.
15. Hilt [Електронний ресурс] – Режим доступу до ресурсу:

https://developer.android.com/training/dependency-injection/hilt-android.

16. Smyth N. Android Studio 3.0 Development Essentials - Android 8 Edition. CreateSpace Independent Publishing Platform, 2017. 726 с.
17. Navigation Component [Електронний ресурс] – Режим доступу до ресурсу: https://developer.android.com/guide/navigation.
18. ANTLR [Електронний ресурс] – Режим доступу до ресурсу: https://www.antlr.org/.