НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ННК "Інститут прикладного системного аналізу"
(повна назва інституту/факультету)

Кафедра Системного проектування
(повна назва кафедри)

«До захисту допущено»

Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)

"___"_____2016 р.

Дипломна робота

першого (бакалаврського) **рівня вищої освіти**
(першого (бакалаврського), другого (магістерського))

зі спеціальності 7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код та назва спеціальності)

на тему: Дослідження функціонально-орієнтованих розширень фреймворків для створення Android застосувань

Виконав (-ла): студент (-ка) __4__ курсу, групи __ДА-22___
(шифр групи)

Сутула Олександр Віталійович _____
(прізвище, ім'я, по батькові) (підпис)

Керівник ас. Голубова І. А. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант Економічна професор, д.е.н. Семенченко Н.В. _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

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

Нормоконтроль ст. викладач Бритов О.А. _____
(посада, науковий ступінь, 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 – 2016 року

Національний технічний університет України
«Київський політехнічний інститут»

Факультет (інститут) ННК "Інститут прикладного системного аналізу"
(повна назва)

Кафедра Системного проектування
(повна назва)

Рівень вищої освіти Перший(Бакалаврський)
(перший (бакалаврський))

Спеціальність 7.05010102, 8.05010102 Інформаційні технології проектування
7.05010103, 8.05010103 Системне проектування
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ А.І.Петренко
(підпис) (ініціали, прізвище)
«___»_____2016 р.

ЗАВДАННЯ
на дипломний проект (роботу) студенту
Сутулі Олександру Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Дослідження функціонально-орієнтованих розширень фреймворків для створення Android застосувань
керівник проекту (роботи) Голубова Ірина Андріївна, ас,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від 12 травня 2016 р. № 50-ст

2. Строк подання студентом проекту (роботи) 08.06.2016

3. Вихідні дані до проекту (роботи) _____

Операційна система Android 4.3
Частота процесору 1.2 ГГц
Середовище розробки - Intelij Idea Community edition
Система збірки проектку Sbt, Gradle

4. Зміст розрахунково-пояснювальної записки (перелік завдань, які потрібно розробити)

1. Дослідження технологій розробки Android застосувань.
2. Аналіз існуючих фреймворків що підтримують функціонально-орієнтовану парадигму.
3. Визначення критеріїв для порівняння функціонально-орієнтованих розширень фреймворків.

4. Проведення тестування по обраним критеріям.

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслеників, плакатів тощо)

1. Порівняння мов по системі типів - плакат;
2. Результати тестування плакат №1 - плакат;
3. Порівняння використання ресурсів у різних середовищах виконання – плакат;
4. Презентація

6. Консультанти розділів проекту (роботи)[·]

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Професор, д.е.н Семенченко Н.В.		

7. Дата видачі завдання 01.02.2016

Календарний план

№ з/п	Назва етапів виконання дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	01.02.2016	
2	Збір інформації	15.02.2016	
3	Дослідження проблеми поточної парадигми програмування для розробки застосувань	28.02.2016	
4	Знаходження критеріїв порівняння різних способів та тестових приклади	10.03.2016	
5	Написання тестових прикладів	15.03.2016	
8	Порівняння результатів дослідження	30.04.2016	
9	Оформлення дипломної роботи	31.05.2016	
10	Отримання допуску до захисту та подача роботи в ДЕК	08.06.2016	

Студент _____ О.В.Сутула
(підпис) (ініціали, прізвище)

Керівник проекту (роботи) _____ І.А.Голубова
(підпис) (ініціали, прізвище)

[·] Консультантом не може бути зазначено керівника дипломного проекту (роботи).

АНОТАЦІЯ

бакалаврської роботи Сутули Олександра Віталійовича

на тему: «Дослідження функціонально-орієнтованих розширень фреймворків для створення Android застосувань»

Дипломна робота присвячена дослідженню існуючих функціонально-орієнтованих розширень фреймворків для створення Android застосувань. Метою роботи є підвищення ефективності створення Android застосувань. У роботі приведений огляд існуючих засобів для мобільної розробки їх переваги та недоліки. Проведено порівняння сучасних можливостей функціонально-орієнтованих розширень, в результаті якого для подальшого детального аналізу обрано мову Scala та фреймворк RxJava. Досліджено принципи та вимоги до технології мобільної розробки та на їх основі визначено критерії оцінки раціональності використання розширень. Проведено тестування, що експериментально доводить доцільність використання функціонально-оріентованих фреймворків у розробці мобільних застосувань для операційної системи Android. Робота була апробована в 2 наукових журналах і одній міжнародній конференції.

Загальний обсяг роботи 81 сторінки, 17 малюнків, 10 таблиць, 18 бібліографічних найменувань.

Перелік ключових слів: функціонально-орієнтоване розширення, Scala, RxJava, мобільна розробка, Android, критерії порівння, фреймворк, мобільні застосування, тестування.

АННОТАЦИЯ

бакалаврской дипломной работе Сутулы Александра Виталиевича

на тему:«Исследование функционально-ориентированных расширений фреймворков для создания Android приложений»

Дипломная работа посвящена исследованию существующих функционально-ориентированных расширений фреймворков для создания Android приложений. Целью работы является повышение эффективности создания Android приложений.

В работе приведен обзор существующих средств для мобильной разработки их преимущества и недостатки. Проведено сравнение современных возможностей функционально-ориентированных расширений, в результате которого для дальнейшего детального анализа выбран язык Scala и фреймворк RxJava. Исследованы принципы и требования к технологии мобильной разработки и на их основе определены критерии оценки рациональности использования расширений. Проведено тестирование, экспериментально доказывает целесообразность использования функционально-ориентированных фреймворков в разработке мобильных приложений для операционной системы Android. Работа была апробирована в 2 научных журналах и одной международной конференции.

Общий объем работы 81 страница, 17 рисунков, 10 таблиц, 18 библиографических наименований.

Перечень ключевых слов: функционально-ориентированное расширение, Scala, RxJava, мобильная разработка, Android, критерии сравнение, фреймворк, мобильные приложения, тестирование.

ANNOTATION

for the bachelors work of Sutula Oleksandr

«The study of functional-oriented extensions of frameworks to create Android applications»

Research paper is devoted to research existing function-oriented extensions of frameworks for creating Android apps. The aim is to increase the efficiency of creating Android apps.

The paper provides an overview of existing tools for mobile development, their advantages and disadvantages. A comparison of the current capabilities of function-oriented extensions, which resulted in the further detailed analysis of the selected language Scala and RxJava framework. Investigated principles and requirements for the development mobile apps and based on them defined criteries for assessing the rationality using extensions. Testing proves the feasibility of using functional-oriented frameworks in the development of mobile applications for the Android operating system. Study was tested in two scientific journals and one international conference.

Total of 81 pages, 17 pictures, 10 tables, 18 bibliographical items.

Keyword list: functionally-oriented extension, Scala, RxJava, mobile development, Android, comparison criteria, framework, mobile application testing.

ЗМІСТ

ВСТУП

Android платформа на сьогоднішній день є найпоширенішою платформою для мобільних пристроїв. 82.8% девайсів усіх можливих конфігурацій працюють на цій операційній системі. 51% медійного часу користувачів припадає на мобільні пристрої. У той же час, з такою широкою аудиторією розробка застосувань має орієнтуватись на середовище яке постійно змінюється та має обмежені ресурси пам’яті, батареї, процесорної обчислювальної здатності, інтернет з’єднання тощо. Мають бути передбачені усі можливі ситуації користування мобільним пристроєм. Отже основною ціллю розробки є створення застосувань для постійного користування в нестабільних умовах що охоплює максимум аудиторії.

Дуже важливою є проблема вибору засобів розробки: мови, фреймворків, бібліотек, тощо. Адже інструменти визначають кількість необхідного часу для створення проекту, зусиль на підтримку, легкість внесення змін, розмір проекту, структуру і т.і.

Сучасна розробка мобільних застосувань для ОС Android передбачає використання Java в якості основної мови програмування.

Java 6, що використовується 80% пристроїв, випущена в 2006 році, що означає відсутність можливості використання сучасних підходів у розробці та моральну застарілість цього інструменту. Застарілість інструменту збільшує час розробки застосування та зменшує його надійність у порівнянні із сучасними мовами та підходами. Отже, для зменшення часу розробки необхідно знайти новий спосіб, який би залишав переваги Java і ліквідовував її недоліки. Серед переваг: JVM, надійність, стабільність, велика база сторонніх бібліотек, можливість використання коду написаного на попередній версії. Недоліки: нелаконічність коду, складність роботи з потоками, відсутність підтримки сучасних можливостей та підходів, обробка помилок.

У наш час дуже актуальною є проблема збільшення продуктивності процесора, тим більше на мобільних пристроях, що є обмеженими в розмірах. Сьогодні ми спостерігаємо факт того, що виробники створюють багатоядерні мобільні пристрої для збільшення їхньої ефективності. Тому гостро стає питання оптимізації застосувань для роботи з декількома процесорами. Натомість Java має механізм синхронізації потоків що підвищує складність розробки програмного забезпечення, а також ризик виникнення помилок та складність підтримки. Тому все більш популярною стає функціональна парадигма, яка вирішує проблему використання багатопочності без синхронізіції ресурсів.

Використання віртуальної машини Java дає можливість альтернативних способів розробки. Таких, наприклад, як функціонально-орієнтовані розширення, що дозволяють використовувати принципи парадигми функціонального програмування.

Способів використання функціональної парадигми для розробки програм для Android, з використанням Dalvik VM два:

1. Використання сторонніх бібліотек.
2. Використання функціональних мов, що компілюються в байт код JVM.

Вони мають свої особливості, кожна з яких впливає на ефективність розробки і реалізацію застосування по-різному.

Наприклад сторонні бібліотеки, за рахунок багатослівності Java, можуть значно збільшити та ускладнити код проекту.

А функціональні мови розроблені й оптимізовані саме для роботи з JVM. Dalvik VM, яка використовується для Android, має певні відмінності як у розрядності системи, так і в роботі з пам’ятю, а отже поведінка таких мов може бути непередбачуваною.

Завданням дипломної роботи є вибір, оцінка і аналіз ряду показників, що допоможуть у подальшому приймати рішення про доцільність застосування інструментів для функціонально-орієнтованих розширень. А також розгляд зміни структури проекту та реалізації програмного коду з їх використанням.

1 ANDROID. ТЕХНОЛОГІЇ РОЗРОБКИ МОБІЛЬНИХ ЗАСТОСУВАНЬ

1.1 Класичний підхід до розробки Android застосувань

Android — операційна система і платформа для мобільних телефонів та планшетних комп'ютерів, створена компанією Google на базі ядра Linux. Android платформа на сьогоднішній день є найпоширенішою платформою для мобільних пристроїв.

Таб. 1.1 – Поширеність мобільних платформ у світі [1]

Період	Android	IOS	Windows Phone	BlackBerry OS	Інші
2015Q2	82.8%	13.9%	2.6%	0.3%	0.4%
2014Q2	84.8%	11.6%	2.5%	0.5%	0.7%

82.8% пристроїв у 2015 році усіх можливих конфігурацій працюють на цій операційній системі. 51% медійного часу користувачів припадає на мобільні пристрої.

Android базується на ядрі Linux. Базовим елементом цієї операційної системи є реалізація Dalvik віртуальної машини Java, і все програмне забезпечення і застосування спираються на цю реалізацію Java.

Крім того в 2009 році на застосунок до ADT був опублікований Android Native Development Kit (NDK)[5], пакет інструментаріїв і бібліотек дозволяє вести розробку застосунків мовою C/C++. NDK рекомендується використовувати для розробки ділянок коду, критичних до швидкості. Але він не пристосований для широкого кола розробників, тому має певні складності у налаштуванні й використанні, також розробка за допомогою цього набору

інструментів має бути написаний під конкретну архітектуру мобільного пристрою.

1.1.1 Особливості Dalvik віртуальної машини

Dalvik оптимізований для низького споживання пам'яті, це не стандартна регістр-орієнтована віртуальна машина, яка добре підходить для виконання на RISC-архітектурах процесорів, котрі часто використовуються у мобільних та вбудованих пристроях, таких, як комунікатори й планшетні комп'ютери.

Програми для Android, як правило, написані на Java і скомпільовані в байт-код для віртуальної машини Java, яка потім транслюється в Dalvik байткод і зберігається в .dex (Dalvik Executable) і .odex (оптимізоване Dalvik Executable) файлах. Компактний формат Dalvik Executable призначений для систем, які обмежені з точки зору пам'яті і швидкості процесора.

На відміну від віртуальних машин Java, які використовують стек машину, Dalvik VM використовує архітектуру на основі регістрів, яка вимагає меншу кількість, але як правило, більш складних, інструкцій віртуальної машини. Програми Dalvik написані на Java з використанням інтерфейсу Android програмування застосувань (API), компілюються в Java байт-код, і перетворюються в інструкції Dalvik по мірі необхідності.

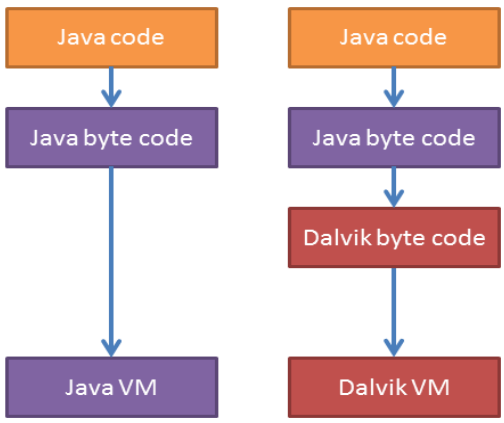

Рисунок 1.1 – Порівняння компіляції в стандартний JVM код і Dalvik VM [3]

Інструмент DX використовується для перетворення Java .class файлів у формат .dex. Кілька класів можуть бути включені в один файл .dex. Повторювані строкові константи та інші, що використовуються в декількох файлах класів включені тільки один раз в вихідний .dex для економії місця. Java-байт-код також може бути перетворений в альтернативний набір інструкцій, який використовується Dalvik VM. Розмір файлу .dex, як правило, на кілька відсотків менше, ніж стиснений архів Java (JAR), отриманий з одних і тих же файлів .class. [2]

Виконувані файли Dalvik можуть бути змінені знову при встановленні на мобільний пристрій. Наприклад для того, щоб отримати додаткову оптимізацію, порядок байт може бути змінений в певних наборах даних, прості структури даних і фукції бібліотек можуть бути пов'язані inline.

Dalvik VM має такі відмінності від стандартних JVM:

- Стандартний Java-байт-код виконує 8-розрядні команди стека. Локальні змінні повинні бути скопійовані в або з стека операндів окремими інструкціями. Dalvik замість цього використовує свій власний 16-бітний набір команд, який працює безпосередньо на локальних змінних. Локальна змінна зазвичай вибрані поля 4-розрядного "віртуального регістру". Це знижує кількість команд Dalvik і підвищує швидкість його інтерпретатора.
- Константний пул був змінений, щоб використовувати тільки 32-розрядні індекси для спрощення інтерпретатора.
- Віртуальна машина розроблена для використання меньшої кількості пам’яті.

За словами Google, дизайн Dalvik дозволяє пристрою ефективно запускати кілька віртуальних машин. [4]

1.1.2 Android NDK

Android NDK (Android Native Development Kit) — необхідний набір інструментарію для розробки компонентів програмнго забезпечення для платформи Android який базується на C/C++ та інших мовах програмування. Містить в собі лімітований набір загальновживаних низькорівневих(нативних) бібліотек та API написаних на C/C++ та інших мовах програмування, документацію і мінімальний набір прикладів для демонстрації базового функціоналу. За допомогою NDK розробник застосування для операційної системи Android може імплементувати окремі його частини використовуючи такі мови як C/C++, а не тільки на Java. Це надає можливість використати деякі переваги, так як в окремих випадках код написаний на C/C++ буде виконуватись швидше в порівнянні з кодом на Java. Android NDK може бути використаний для платформи Android 1.5(API Level 3) і більш нових версій.

Враховуючи відмінності які існують між кодом написанным на C/C++ в порівнянні з кодом написаним на Java, розробниками Google рекомендовано використовувати Android NDK в наступних цілях:

- пришвидшення розрахунків великих обчислень, таких як обробка сигналів, розрахунки для фізичних симуляцій, сортування та інші.
- використання функціоналу із сторонніх бібліотек написаних на C/C++, наприклад: OpenCV, OpenSL ES.
- програмування на низькому рівні, або у випадках коли Java не надає необхідного інструментарію.

1.1.3 Java

Java – об’єктно-орієнтована мова програмування зі статичною типізацією яка заснована на імперативній парадигмі програмування. Код компілюється в байт-код і працює на Java Virtual Machine. Це згладжує, як недоліки, що виникають при чистій компіляції так і недоліки, що виникають при чистій

інтерпретації. Принципи Java – простота мови, безпека і можливість перенесення коду. Java достатньо надійний інструмент що існує вже більше 20 років, має найбільший арсенал бібліотек та інструментів із існуючих мов програмування.

У створенні мови програмування Java було чотири початкові цілі:

- Синтаксис мови повинен бути «простим, об'єктно-орієнтовним та звичним».
- Реалізація має бути «безвідмовною та безпечною», а також «високопродуктивною».
- Повинна зберегтися «незалежність від архітектури та портативність».
- Мова має бути «динамічною, інтерпретованою та підтримувати мультиопрацьовування».

Під «незалежністю від архітектури» мається на увазі те, що програма, написана на мові Java, працюватиме на будь-якій підтримуваній апаратній чи системній платформі без змін у початковому коді та перекомпіляції.

Цього можна досягти, компілюючи початковий Java код у байт-код, який являє собою спрощені машинні команди. Потім програму можна виконати на будь-якій платформі, що має встановлену віртуальну машину Java, яка інтерпретує байткод у код, пристосований до специфіки конкретної операційної системи і процесора. Зараз віртуальні машини Java існують для більшості процесорів і операційних систем.

Стандартні бібліотеки забезпечують загальний спосіб доступу до таких платформозалежних особливостей, як обробка графіки, багатопотоковість та роботу з мережами. У деяких версіях задля збільшення продуктивності JVM байт-код можна компілювати у машинний код до або під час виконання програми.

Основна перевага використання байт-коду — це портативність. Тим не менш, додаткові витрати на інтерпретацію означають, що інтерпретовані програми будуть майже завжди працювати повільніше, ніж скомпільовані у машинний код, і саме тому Java одержала репутацію «повільної» мови. Проте, цей розрив суттєво скоротився після введення декількох методів оптимізації у сучасних реалізаціях JVM.

Одним із таких методів є компіляція just-in-time (JIT, що перетворює байт-код Java у машинний під час першого запуску програми, а потім кешує його. У результаті така програма запускається і виконується швидше, ніж простий інтерпретований код, але ціною додаткових витрат на компіляцію під час виконання. Складніші віртуальні машини також використовують динамічну рекомпіляцію, яка полягає в тому, що В. М. аналізує поведінку запущеної програми й вибірково рекомпілює та оптимізує певні її частини. З використанням динамічної рекомпіляції можна досягти більшого рівня оптимізації, ніж за статичної компіляції, оскільки динамічний компілятор може робити оптимізації на базі знань про довкілля періоду виконання та про завантажені класи. До того ж він може виявляти так звані гарячі точки (англ. hot spots) —- частини програми, найчастіше внутрішні цикли, які займають найбільше часу при виконанні. JIT-компіляція та динамічна рекомпіляція збільшує швидкість Java-програм, не втрачаючи при цьому портативності.

Існує ще одна технологія оптимізації байткоду, широко відома як статична компіляція, або компіляція ahead-of-time (AOT). Цей метод передбачає, як і традиційні компілятори, безпосередню компіляцію у машинний код. Це забезпечує хороші показники в порівнянні з інтерпретацією, але за рахунок втрати переносності: скомпільовану таким способом програму можна запустити тільки на одній, цільовій платформі.

Швидкість офіційної віртуальної машини Java значно покращилася з моменту випуску ранніх версій, до того ж, деякі випробування показали, що

продуктивність JIT-компіляторів у порівнянні зі звичайними компіляторами у машинний код майже однакова. Проте ефективність компіляторів не завжди свідчить про швидкість виконання скомпільованого коду, тільки ретельне тестування може виявити справжню ефективність у даній системі.

Але Java має також ряд недоліків:

1. Багатослівність та об’ємність коду. Проекти написані на Java трудоємкі в плані розширення, адже для мінімальної функціональності необхідно виконати ряд рутинних дій, а також турбуватись про архітектуру застосування кожного разу при додаванні нової функціональності. Це є причиною того що проекти стають дуже важкими для осмислення не тільки новачку, а й тому хто давно в проекті. Чим більше коду – тим більша імовірність помилки. Для обробки помилок необхідно також створювати низку об’єктів та перевірок.
2. Зворотня підтримка. Кожна нова версія мови має підтримувати код написаний за допомогою попередньої. Звісно, це дає можливість використання коду попередніх версій, але через цю умову мова дуже повільно розвивається і деякі помилки що були зроблені на початку її проектування тягнуться через усі версії.
3. Складність обробки потоків. Через використання імперативної парадигми, виникає проблема синхронізації доступних ресурсів між двома потоками. Тому зазвичай багатопоточність ускладнюється і є небезпечною та погано передбачуваною.

1.2 Особливості та переваги функціональної парадигми

Парадигма програмування — це система ідей і понять, які визначають стиль написання комп'ютерних програм, а також спосіб мислення програміста.

Парадигма програмування — спосіб концептуалізації, що визначає організацію обчислень і структурування роботи, яку виконує комп'ютер. [6]

Парадигма програмування унаочнює те, як програміст розглядає роботу програми; наприклад, за ООП — як множини об'єктів, тоді як за ФП — як послідовності обчислень функцій без станів. [7]

Кожну окрему парадигму програмування характеризує наявність у ній метода та зв'язок із моделлю життєвого циклу. Спільним для різних парадигм програмування є загальні принципи проектування програмного продукту. Користувач може вибирати ту або іншу парадигму програмування з позицій зручності застосування для задач у ПрО та виготовлення конкретного програмного продукту. [8]

Розглядають наступні основні парадигми програмування. Існує неминуче дублювання цих парадигм, основні риси або визачальні відмінності наведені нижче:

Імперативне програмування - визначає обчислення як оператори, які змінюють стан програми

Функціональне програмування - розглядає обчислення як оцінки математичних функцій і уникає стану і змінюваних даних

Об'єктно-орієнтоване програмування (ООП) - організовує програми в якості об'єктів: структур даних, що складаються з полів даних і методів разом з їх взаємодій.

Реактивне програмування - потік програми визначається подіями, такими як виходи датчиків або дій користувача (клацань миші, натискань на клавіші) або повідомлень від інших програм або ниток.

1.2.1 Імперативне програмування

Імперативне програмування — парадигма програмування, згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми. Подібно до того, як з допомогою наказового способу в

мовознавстві перелічується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп'ютера для виконання.

Апаратна реалізація практично всіх комп'ютерів є імперативною; майже все комп'ютерне обладнання призначене для виконання машинного коду, який є рідним для комп'ютера, написаному в імперативному стилі. З точки зору низького рівня, програмний стан визначається вмістом пам’яті, а вирази – інструкції на рідній машинній мові комп’ютера. Імперативні мови високого рівня використовують змінні і більш складні оператори, але як і раніше слідують тій же парадигмі. Кожен крок є інструкцією, і фізичний світ зберігає стан. Так як основні ідеї імперативного програмування є концептуально знайомі і безпосередньо втілені в апаратних засобах, більшість комп'ютерних мов в імперативному стилі.

Оператори присвоювання, в імперативній парадигмі, виконують операцію над інформацією, що знаходиться в пам'яті і зберігає результати в пам'яті для подальшого використання. Імперативні мови високого рівня, крім того, дозволяють оцінити складні вирази, які можуть складатися з комбінації арифметичних операцій і значень функцій, а також привласнення результуючого значення в пам'яті. Оператори циклів дозволяють послідовність інструкцій виконати кілька разів. Цикли можуть або виконувати оператори вони містять задане число раз, або вони не можуть виконувати їх до тих пір, деякі зміни стану. Оператори умови дозволяють послідовність операторів, яка буде виконана тільки при виконанні деякої умови. В іншому випадку, оператори пропускаються і послідовність виконання триває від оператора після їх. Безумовні розгалужені заяви дозволяють послідовність виконання повинні бути передані в якійсь іншій частині програми. До них відносяться стрибок (так званий "Goto" на багатьох мовах), перемикач і підпрограми або процедури, виклик (який зазвичай повертається до наступного оператору після виклику).

1.2.2 Функціональне програмування

Функціональне програмування — парадигма програмування, яка розглядає програму як обчислення математичних функцій та уникає станів та змінних даних. Функціональне програмування наголошує на застосуванні функцій, на відміну від імперативного програмування, яке наголошує на змінах в стані та виконанні послідовностей команд.

Іншими словами, функціональне програмування є способом створення програм, в яких єдиною дією є виклик функції, єдиним способом розбиття програми є створення нового імені функції та задання для цього імені виразу, що обчислює значення функції, а єдиним правилом композиції є оператор суперпозиції функцій. Жодних комірок пам'яті, операторів присвоєння, циклів, ні, тим більше, блок-схем чи передачі управління.[9]

Ширша концепція функційного програмування визначає набір спільних правил та тем замість переліку відмінностей від інших парадигм. До таких, що часто вважаються важливими, належать функції вищого порядку[10] та функції першого класу: замикання, та рекурсія. До інших поширених можливостей функційних мов програмування належать продовження, система типізації Хіндлі-Мілнера, нечіткі обчислення (включно з, але, не обмежуючись, лінивими обчисленнями), та монади.

Основну увагу функціональним мовам програмування, особливо, «чисто функціональним», приділили академічні дослідники. Однак, до відомих функціональних мов програмування, які використовуються в промисловості та комерційному програмуванні належить Erlang (паралельні програми), R (статистика), Mathematica (символьні обчислення), J та K (фінансовий аналіз), та спеціалізовані мови програмування наприклад XSLT. Істотний вплив на функціональне програмування здійснило лямбда-числення, мова програмування APL, мова програмування Lisp та новіша мова програмування Haskell.

Основною особливістю функціонального програмування, яка визначає як переваги, так і недоліки даної парадигми, є те, що в ній реалізується модель обчислень без станів. Якщо імперативна програма на будь-якому етапі виконання має стан, тобто сукупність значень всіх змінних, і виробляє побічні ефекти, то чисто функціональна програма ні цілком, ні частинами стану не має і побічних ефектів не виробляє. Те, що в імперативних мовах робиться шляхом присвоювання значень змінним, в функціональних досягається шляхом передачі виразів в параметри функцій. Безпосереднім наслідком стає те, що чисто функціональна програма не може змінювати вже наявні в ній дані, а може лише породжувати нові, шляхом копіювання та / або розширення старих. Наслідком того ж є відмова від циклів на користь рекурсії.

Принципово немає перешкод для написання програм у функціональному стилі на мовах, які традиційно не вважаються функціональними, точно так само, як програми в об'єктно-орієнтованому стилі можна писати на структурних мовах. Деякі імперативні мови підтримують типові для функціональних мов конструкції, такі як функції вищого порядку і спискові включення (list comprehensions), що полегшує використання функціонального стилю в цих мовах. Прикладом може бути програмування на мові Python.

У мові C покажчики на функцію в якості типів аргументів можуть бути використані для створення функцій вищого порядку. Функції вищого порядку і відкладена списковому структура реалізовані в бібліотеках C++. У мові C # версії 3.0 і вище можна використовувати λ-функції для написання програми в функціональному стилі. У складних мовах, типу Алгол-68, наявні засоби метапрограмування (фактично, доповнення мови новими конструкціями) дозволяють створити специфічні для функціонального стилю об'єкти даних і програмні конструкції, після чого можна писати функціональні програми з їх використанням.

Сильні сторони:

- Підвищення надійності коду за рахунок чіткої структуризації та відсутності необхідності відстеження побічних ефектів. Будь-яка функція працює тільки з локальними даними і працює з ними завжди однаково, незалежно від того, де, як і за яких обставин вона викликається. Неможливість мутації даних при користуванні ними в різних місцях програми виключає появу помилок, які тяжко виявити (таких, наприклад, як випадкове присвоювання невірного значення глобальній змінній в імперативній програмі).
- Оскільки функція у функціональному програмуванні не може породжувати побічні ефекти, змінювати об'єкти не можна як усередині області видимості, так і зовні (на відміну від імперативних програм, де одна функція може встановити яку-небудь зовнішню змінну, що зчитується іншою функцією). Єдиним ефектом від обчислення функції є повернений їй результат, і єдиний чинник, який впливає на результат — це значення аргументів.
- Таким чином, є можливість протестувати кожну функцію в програмі, просто обчисливши її від різних наборів значень аргументів. При цьому можна не турбуватися ні про виклик функцій в правильному порядку, ні про правильне формуванні зовнішнього стану. Якщо будь-яка функція в програмі проходить модульні тести, то можна бути впевненим у якості всієї програми. В імперативних програмах перевірка значення, що повертається функції, недостатня: функція може модифікувати зовнішній стан, який теж потрібно перевіряти, чого не потрібно робити в функціональних програмах.
- Функціональне програмування дозволяє описувати програму в так званому «декларативному» вигляді, коли жорстка послідовність виконання багатьох операцій, необхідних для обчислення результату, в явному вигляді не задається, а формується автоматично в процесі

обчислення функцій. Ця обставина, а також відсутність станів дає можливість застосовувати до функціональних програм досить складні методи автоматичної оптимізації.

- Ще однією перевагою функціональних програм є те, що вони надають найширші можливості для автоматичного розпаралелювання обчислень. Оскільки відсутність побічних ефектів гарантовано, в будь-якому виклику функції завжди припустиме паралельне обчислення двох різних параметрів — порядок їх обчислення не може вплинути на результат виклику.

Недоліки:

- Відсутність присвоювання і заміна їх на породження нових даних призводять до необхідності постійного виділення та автоматичного звільнення пам'яті, тому в системі виконання функціональної програми обов'язковим компонентом стає високоефективний збирач сміття.
- Нестрога модель обчислень призводить до непередбачуваного порядку виклику функцій, що створює проблеми при введенні-виведенні, де порядок виконання операцій є важливим. Крім того, очевидно, функції введення в своєму природному вигляді (наприклад, getchar із стандартної бібліотеки мови C) не є чистими, оскільки здатні повертати різні значення для одних і тих же аргументів, і для усунення цього потрібні певні хитрощі.

При використанні віртуальної машини Java високоефективний та перевірений збирач сміття вже вбудований. Через властивість незмінності об’єктів, для проміжних задач ми використовуємо недовгострокові нові об’єкти, що дозволяє більш ефективно працювати збирачу сміття віртуальної машини Java.

Також ця парадигма має ряд нових структур, абстракцій та операторів, які дозволяють ефективніше описувати вирішення проблеми. Серед них:

- функції вищих порядків;
- чисті функції;
- монади.

Функція вищого порядку — це функція, що приймає в якості аргументів інші функції або повертає іншу функцію в якості результату. Основна ідея полягає в тому, що функції мають той же статус, що й інші об'єкти даних.

Наприклад, функція вищого порядку може бути використана для реалізації незмінної частини алгоритму, у той час як змінна частина має бути реалізована у функції, що передається аргументом. Типовим прикладом може бути функція сортування данних. Сортування данних потребує порівняння екземплярів данних, яке може бути різним для різних типів данних. Без використання функції вищого порядку було б необхідно створювати окрему функцію сортування для кожного типу данних. Але ж, зазвичай, сам алгоритм сортування не залежить від алгоритму порівняння й залежить ліше від результату цього порівняння. Можливість вказати, як аргумент виклику, яку саме функцію порівняння треба використовувати для цього виклику, дає можливість створити універсальну функцію сортування. У той же час, та ж сама функція порівняння може бути використона як аргумент й для інших функцій вищого порядку, наприклад, для функцій пошуку максимального чи мінімального елементу.

Використання функцій вищого порядку призводить до абстрактних і компактних програм, беручи до уваги складність вироблених ними обчислень[11]. Іноді функції вищого порядку називають функторами хоча це не зовсім правильно, з математичної точки зору вони є функціоналами чи операторами.

У функціональних мовах програмування всі функції, що приймають декілька аргументів, є функціями вищого порядку, оскільки вони є

каррінгованими, що дає можливість наче частково виконати функцію, в результаті чого отримати нову функцію вже від меншого числа аргументів.

Чистими називають функції, які не мають побічних ефектів вводу-виводу і пам'яті (вони залежать тільки від своїх параметрів і повертають тільки свій результат). Чисті функції володіють декількома корисними властивостями, багато з яких можна використовувати для оптимізації коду:

Якщо результат чистої функції не використовується, він може бути видалений без шкоди для інших виразів.

Результат виклику чистої функції може бути мемоізований, тобто збережений в таблиці значень разом з аргументами виклику. Якщо надалі функція викликається з цими ж аргументами, її результат може бути взятий прямо з таблиці, не обчислюючи її знову (іноді це називається принципом прозорості посилань). Мемоізація, ціною невеликої затрати пам'яті, дозволяє істотно збільшити продуктивність і зменшити порядок зростання деяких рекурсивних алгоритмів.

Якщо немає ніякої залежності серед даних між двома чистими функціями, то порядок їх обчислення можна поміняти (інакше кажучи обчислення чистих функцій задовольняє принципам thread-safe)

Якщо вся мова не допускає побічних ефектів, то можна використовувати будь-яку політику обчислення. Це надає свободу компілятору комбінувати і реорганізовувати обчислення виразів у програмі (наприклад, виключити деревоподібні структури).

Хоча більшість компіляторів імперативних мов програмування розпізнають чисті функції і видаляють загальні підвирази для викликів чистих функцій, вони не можуть робити це завжди для попередньо скомпільованих бібліотек, які, як правило, не надають цю інформацію. Деякі компілятори, такі

як gcc, з метою оптимізації надають програмісту ключові слова для позначення чистих функцій[10]. Fortran 95 дозволяє позначати функції як «pure»[11].

Монада - це абстракція лінійного ланцюжка пов'язаних обчислень. Її основне призначення - інкапсуляція функцій з побічним ефектом від чистих функцій, а точніше їх виконань від обчислень.

1.2.3 Реактивне програмування

Реактивне програмування - парадигма програмування, орієнтована на потоки даних і росповсюдження змін.

Реактивне програмування пропонується як шлях легкого створення користувацьких інтерфейсів, анімації, або моделювання систем, що змінюються в часі. Дана парадигма базується на чотирьох принципах: відмовостійкості, чутливості до змін, гнучкості, маштабування. Наприклад, у MVC архітектурі за допомогою реактивного програмування можна реалізувати автоматичне відображення зміни з Model у View, і навпаки, що значно покращує якість коду, роблячи його більш коректним, а також зменшує його об’єм.

Більшість Android застосувань побудовані на постійній взаємодії з користувачем, що породжує необхідність впровадження підходів реактивного програмування та функціональної парадигми в процес розробки. Це призведе до збільшення прозорості роботи застосування, полегшення читання коду, уникнення помилок пов’язаних зі станом програми, ліквідування.

1.3 Список мов JVM

JVM була спочатку створена для підтримки виключно мови програмування Java. Однак, з плином часу, деякі мови були адаптовані або створені для виконання на платформі Java.

Цей список мов є зібранням мов програмування, які призначені для створення програмного забезпечення, що використовує в якості середовища

виконання віртуальну машину Java (JVM) та орієнтовані на функціональну парадигму програмування. Деякі з цих мов інтерпретуються, а деякі компілюються в байт-код Java і компілюються «на льоту» під час виконання.

- Clojure - функціональна мова, діалект Lisp;
- Groovy – мова сценаріїв;
- Scala - об'єктно-орієнтована і функціональна мова.

Таб. 1.2 – Порівняння мов по системі типів

Мова	Безпека типів	Типізація	Приведенння типів	Перевірка типів
C++	небезпечна	явна	номінальне	статична
Clojure	безпечна	неявна з можливістю явного назначення типів		динамічна
Groovy	безпечна	неявна з можливістю явного назначення типів		динамічна з можливістю статичності
Java	безпечна	явна	номінальне	статична
Scala	безпечна	частково неявна	номінальне та структурне	статична

Визначення

Номінальний тип системи означає, що мова розглядає доцільність внесення типи є сумісними і / або еквівалент на основі явних декларацій і імен.

Структурний тип системи означає, що мова розглядає доцільність внесення типи сумісні і / або його еквівалент на основі визначення і характеристики типів.

Перевірка типів визначає, чи буде і коли перевіряються типи. Статична перевірка означає, що помилки типу представлені на основі тексту (вихідний код) або іншої програми. Динамічна перевірка означає, що помилки типу представлені на основі динамічна програми (час виконання) поведінки.

Таб. 1.3 – Порівняння мов по підтримці парадигм

Мова	Імперативна	ООП	Функціональна	Процедурна	Загальна	Рефлективна
Clojure			+			
C++	+	+	+	+	+	
Groovy	+	+	+			
Java	+	+		+	+	+
Scala	+	+	+	+	+	+

Як бачимо з таблиці 1.3, найгірше з підтримкою парадигм у Clojure, найкраще у Scala. Підтримка парадигм дає змогу оцінювати мову з точки зору гнучкості та зручності використання з різними інструментами.

1.3.1 Clojure

Clojure — сучасний діалект мови програмування Lisp. Це мова загального призначення, що підтримує інтерактивну розробку, зорієнтовану на функціональне програмування, спрощує багатотредове програмування, та містить риси сучасних скриптових мов.

Clojure працює на Java Virtual Machine і Common Language Runtime. Як і інші Lisp-подібні мови, Clojure розглядає код як дані і має потужну систему макросів.

Clojure - сучасний Lisp для функціонального програмування, розрахований на інтеграцію з розповсюдженою платформою Java й розроблений для паралельного програмування.

Підхід Clojure до паралельності характеризується концепцією тотожностей, що представляють серію незмінних станів протягом часу. Оскільки стани є незмінними значеннями, будь-яка кількість обробників може паралельно обробляти їх, і конкуренція зводиться до питання керування змінами від одного стану до іншого. З цією метою, Clojure надає декілька типів

змінюваних посилань, кожен з яких має добре визначену семантику переходу між станами.

Як і інші Lisp-подібні мови, синтаксис Clojure побудовано на S-виразах (англ. S-expression), які в процесі синтаксичного розбору спершу перетворюються на структури даних за допомогою функції-читача (англ. reader), перш ніж компілюються. Clojure's reader підтримує літеральний синтаксис для хеш-таблиць, множин та векторів на додаток до списків, і вони передаються компілятору як є. Іншими словами, компілятор Clojure компілює не лише спискові структури даних, але й безпосередньо підтримує всі названі вище типи. Clojure — Lisp-1, і не є сумісним за кодом з іншими діалектами мови Lisp.

Система макросів Clojure дуже схожа на використовувану в Common Lisp, з тією відмінністю, що версія синтаксичного цитування Clojure (з використанням знаку `) доповнює символи їхніми просторами імен. Це допомагає запобігти ненавмисному перехопленню імен, оскільки прив'язка до імен, доповнених простором імен, заборонена. Є можливість форсувати таке захоплення імен, але це має бути зроблено явно. Clojure також не дозволяє переприв'язку глобальних імен з інших просторів імен, які були імпортовані в поточний простір.

Можливості мови:

- Компільована мова, що генерує байткод для JVM;
- Тісна інтеграція з Java: відкомпільовані в байткод JVM, програми на Clojure можуть пакуватися та запускатися на JVM-серверах без додаткових ускладнень. Мова також надає макроси, які полегшують використання існуючих Java API. Всі структури даних Clojure реалізують стандартні інтерфейси Java, що робить простим запуск з Java коду, розробленого на Clojure.;
- Динамічна розробка з використанням REPL;

- Функції як об'єкти першого класу;
- Наголос на рекурсії замість циклів з побічним ефектом;
- Ліниві послідовності;
- Надає широкий набір незмінюваних персистентних структур даних;
- Паралельне програмування з використанням software transactional memory, система агентів, система динамічних змінних;
- Мультиметоди (аналог перезавантажуваних (англ. overloading) функцій), що підтримують динамічний вибір метода за типами та значеннями довільного набору аргументів (пор. звичайний об'єктно-орієнтований поліморфізм, де вибір здійснюється за типом (фактично) першого аргумента).

Clojure підтримує розробку лише за допомогою функціональної парадигми, що значно ускладнює розробку мобільних застосувань, адже Android SDK має API розроблене для роботи з Java, що використовує імперативну парадигму.

Отже, хоча Clojure й може використовувати Java код та підтримує функціональну парадигму, для розробки мобільних застосувань для Android як аналог основної мови її розглядати не можна.

1.3.2 Groovy

Groovy — об'єктно-орієнтовна динамічна мова програмування, що працює в середовищі JRE. Мова Groovy запозичила деякі корисні якості Ruby, Haskell і Python, але створена для роботи всередині віртуальної машини Java (JVM) і підтримує тісну інтеграцію з Java програмами.

Ключові особливості:

- Функціональна спрямованість
- Режим статичної компіляції для забезпечення підвищеної продуктивності для критичних до швидкості виконання ділянок коду.

Groovy є більш високорівневою мовою програмування порівняно з Java, а отже розробка на ньому зазвичай відбувається швидше. Цьому сприяють перш за все динамічна природа мови, а по-друге, наявні елементи функціональнального програмування, зокрема замикання.

Приймаючи рішення про те, чи варто використовувати її у якомусь конкретному випадку потрібно пам’ятати про динамічну спрямованність мови і використовувати там, де потрібно використовувати саме динамічні мови. Там де потрібна надійність або значна швидкодія рекомендується використовувати статичні мови, зокрема Java чи Scala. Адже відомо, що зробити помилку при розробці в першому випадку значно легше.

Динамічна типізація цієї мови не дозволяє зробити перевірку типів “на льоту”, що ускладнює роботу з Android API через відсутність перевірки типів на етапі компіляції, робить застосування більш нестійкими, через можливість неочікуваної помилки.

1.3.3 Scala

Scala — мультипарадигмова мова програмування, що поєднує властивості об'єктно-орієнтованого та функціонального програмування.

Scala сумісна із існуючими програмами мовою Java, тобто код Scala може викликатися із Java-програм і навпаки. Починаючи з версії 2.11 Scala потребує принаймні Java 6. У той же час, Scala має набагато більше можливостей у версії 2.11, аніж Java 8.

Ключові особливості:

- Лаконічність та гнучкість синтаксису
- Уніфікована та оновлена система типів
- Повна підтримка функціональної парадигми
- Статична типізація

Scala – статично типізована мова, на відміну від Groovy та Clojure. Це робить систему типів більш складною для розуміння, але дозволяє виявити всі помилки на етапі компіляції, а також дозволяє виконувати код швидше. На відміну від цього, динамічний код вимагає більше тестування, щоб гарантувати коректність програми і, як правило, повільніший, для того, щоб забезпечити більшу гнучкість програмування і простоту. Scala, є кращим вибором, коли ефективність виконання, а також впевненість у коректності програми дуже важливі.

Що стосується парадигм програмування, Scala успадковує об'єктно-орієнтовану модель Java і розширює її різними способами. Groovy більшою мірою орієнтований на скорочення багатослівності. У Clojure, об'єктно-орієнтоване програмування замінено функціональним програмування. Scala також має багато функціональних об'єктів програмування, в тому числі функцій, доступних в розширених функціональних мов, як Haskell, і намагається бути агностиком між двома парадигмами, дозволяючи розробнику вибирати між двома парадигмами або, більш часто, деякі їх комбінації.

Що стосується синтаксису подібності з Java, Scala успадковує багато синтаксису Java, так само як і Groovy. Clojure з іншого боку, наслідковує синтаксис Lisp, який відрізняється як зовнішнім виглядом і філософією. Проте, вивчення Scala також вважається важким через її численні додаткові функції.

Отже, вважаючи на особливості мов, а саме: функціональність, схожість із Java, типізацію. А також можливістю використовувати всю попередню кодову базу. Найкращим вибором серед мов, для дослідження саме розширення, стане поліпарадигменна, статичнотипізована Scala, адже вона найбільш схожа на Java, виправляє її недоліки і додає значну кількість корисної функціональності.

1.4 Розширення Java, за допомогою бібліотек та фреймворків

Для того, щоб додати в мову Java певну можливість писати функціональний чи реактивний код існує ряд бібліотек та фреймворків.

1. Functional Java
2. Retrolambda
3. RxJava

Для додавання монад, можна скористатись фреймворком Functional Java. Розширення відбувається за рахунок додавання нових конструкцій, що не наслідуються від стандартних контейнерів, а отже для її використання необхідно переписувати значну частину вже існуючої кодової бази. Також її розширення неповне без бібліотеки Retrolambda, адже без неї ламбди недоступні за допомогою Java 7.

Необхідно зазначити, що Retrolambda не є стабільною на Android, як і Functional Java, адже вони оптимізовані для роботи з JVM, не Dalvik VM.

RXJava - написаний на Scala фреймворк, для реактивного та функціонального кодування. Цей фреймворк є стабільним на Dalvik VM, а також дозволяє використовувати разом з ним такі бібліотеки та фреймворки як:

- Akka Futures
- HTTP-OK

Також він дає можливість позбутись проблем з обробкою послідовних запитів до серверу.

Також його можна використовувати з усіма мовами JVM, що робить незамінним при написанні застосування за допомогою функціонального та

реактивного стилів. Спрощує обробку запитів, роботу з потоками, подіями, обробку потоків тощо.

Отже, з усіх можливих фреймворків для функціонального розширення Java, ми можемо стабільно використовувати лише один - RXJava. Також цей фреймворк доступний для використання з інших мов JVM. Що робить його необхідним застосуванням при використанні функціональної парадигми під час написання Android застосувань.

1.5 Висновок

Отже існує два базових способи розробки мобільних застосувань для платформи Android:

- за допомогою Java, на базі Dalvik VM;
- за допомогою C/C++ за допомогою Android NDK.

Перший спосіб найбільш поширений, адже менш трудоємкий та стабільний відносно Android NDK. Саме він рекомендований компанією Google (основним розробником Android SDK).

Використання Java призводить до ряду проблем, основними з яких є: великий обсяг коду, складність роботи з потоками, а також застарілість засобів розробки, через використання Java 6 у Dalvik VM.

Вирішення проблем із об’ємністю та заплутаністю коду, а також застарілими засобами розробки, можна вирішити за допомогою двох варіантів:

- розширення можливостей Java за допомогою сторонніх бібліотек та фреймворків;
- використання мов що працюють на базі JVM.

Перший спосіб несе в собі ряд переваг, таких як: зручне додавання в проект та повна функціональна підтримка IDE. Але необхідно зазначити, що

проблему нелаконічності коду це не вирішує у випадках коли сторонній інструмент має загальне застосування, або використовує змінну поведінку.

Другий спосіб залишає можливість використовувати всі переваги та засоби іншої мови, комбінуючи з перевагами Java, а також вже написаною кодовою базою. Також ці мови більш продумані з точки зору логіки, та ліквідовують фундаментальні проблеми Java. Використовуючи мови що працюють на базі JVM , не обов’язково відмовлятись від фреймворків, адже ці мови мають можливість використовувати всі переваги Java. Але цей спосіб не такий популярний, та потребує значних трудозатрат у налаштуванні проекту, а також не має повної підтримки IDE.

Проблему роботи з потоками можна вирішити за допомогою використання функціональної парадигми, яка дозволяє легко портувати програмний продукт на роботу з багатьма потоками.

Більшість Android застосувань побудовані на постійній взаємодії з користувачем, що породжує необхідність впровадження підходів реактивного програмування та функціональної парадигми в процес розробки. Використання реактивного програмування в поєднанні з функціональним, дозволяє покращити структуру коду, полегшити створення користувацьких інтерфейсів, а також вирішити проблему з синхронізацією потоків.

Отже, функціонально-орієнтоване розширення Java для Android розробки можливе за допомогою використання іншої мови JVM функціонального призначення, або з використанням фреймворку RXJava.

Мови JVM, що дозволяють функціонально-орієнтоване розширення: Clojure, Groovy та Scala. Clojure - Lisp подібна мова, що ускладнює поєднання її коду з раніше написаним на Java, а також ускладнює взаємодію з елементами стандартного користувацького інтерфейсу. Groovy - динамічно типізована мова, ця особливість дає ряд недоліків, а саме: відсутність виявлення помилок з

типами на етапі попередньої компіляції, уповільнення процесу компіляції. Динамічна типізація в даному випадку неприпустима, адже відбувається компіляція в Java, а потім у Dalvik байт код, що збільшує час створення apk файлу до 2-3 хв. Єдиним припустимим варіантом залишається Scala, яка в своєму арсеналі має всі можливості Java без виключень, а також додає можливість використання функціональної парадигми. Зважаючи на те, що RXJava доступний на всіх мовах JVM, резонно використовувати його і на Java, і на Scala, для нашого дослідження.

2 ПОРІВНЯННЯ ФУНКЦІОНАЛЬНО-ОРІЄНТОВАНИХ РОЗШИРЕНЬ

2.1 Визначення критеріїв порівняння

Аналіз та порівняння двох мов було здійснено на основі таких критеріїв:

- об’єм коду;
- трудозатратність;
- час виконання;
- споживання енергії;
- використання оперативної пам’яті;
- розмір застосування;
- час запуску застосування.

Для визначення раціональності заміни Java на Scala із семантичної точки зору необхідно проаналізувати чи дійсно аналогічний проект на Scala має меньший об’єм коду, а також чи на його написання витрачається стільки ж зусиль.

Отже, для порівняння на основі цих двох критеріїв обрано проект що спочатку був написаний за допомогою Java, а потім переписаний на Scala + RxJava.

Проект є стандартним клієнт-серверним застосуванням з авторизацією, що отримує, відсилає, кешує та зберігає інформацію. Також додаток відправляє на сервер кілька послідовних REST запитів, кожен наступний використовує інформацію попереднього. Ці проекти написані за допомогою стандартного набору бібліотек для клієнт-серверного застосування, серед яких: Retrofit - для роботи з сервером, GSON - для серіалізації об’єктів, бібліотека material design, що дозволяє використовувати сучасні елементи дизайну Android, Picasso - для

відтворення картинок. Бібліотеки обрані з розрахунку збереження для програміста стандартних інструментів з великою кодовою базою. Для всіх проектів було використано спільні XML файли, що описують розмітку. Єдина різниця в коді логіки роботи застосування.

Для оцінки зусиль витрачених при написанні цих проектів було взято час виконання типових завдань протягом розробки проектів. А саме сума таких показників як: створення коду для активності (ініціалізація елементів, логіка взаємодії з користувачем), написання бізнес логіки, створення моделей, виправлення помилок, пошук інформації в інтернеті. Ці показники вимірювались кількістю завдань виконаних кожного тижня. Завдання в даному випадку є одна чітко визначена частина логіки, наприклад авторизація чи форма для відновлення паролю. Виміри при написанні коду Scala з урахуванням вивчення програмістом нової мови.

Отже для того, щоб проаналізувати відмінність у роботі застосування на мобільному пристрої обрано такі критерії: час виконання, споживання енергії, використання пам'яті, розмір програми, час запуску.

Для цих порівняння відмінності у роботі застосування на мобільному пристрої необхідні контрольні завдання які мають бути легко відтворюваними, їх можна легко перевірити, а також вони мають бути сертифікованими фахівцями. ля тесту двох мов було вибрано дві контрольні задачі опубліковані на сайті Computer Language Benchmark Game. Відповідно мовою Java та Scala.

Перша контрольна задача - алгоритм розрахунку гравітаційної проблеми N тіл.

Гравітаційна проблема N тіл є класичною проблемою небесної механіки і гравітаційної динаміки Ньютона.

Вона формулюється в такий спосіб.

У порожнечі знаходиться N матеріальних точок, маси яких відомі {mi}. Нехай попарна взаємодія точок підпорядкована закону тяжіння Ньютона, і нехай гравітації сили адитивні. Нехай відомі початкові на момент часу t = 0 положення і швидкості кожної точки ri | t = 0 = ri0, vi | t = 0 = vi0. Потрібно знайти положення точок для всіх наступних моментів часу.

Ця задача орієнтована на дослідження роботи застосування з важкими обчисленнями та використання процесорної потужності.

Друга контрольна задача - робота з бінарними деревами. Створюється 3 бінарних дерева, вони заповнюються, виконується обхід кожного і видалення.

Ця задача орієнтована на дослідження роботи з пам’яттю, а також роботу збирача сміття.

Суть застосування для тестування використання системних ресурсів у тому, щоб виміряти час виконання завдання. Тобто кожна контрольна задача виконується 10 разів, після чого повідомляється час виконання кожної.

Для виміру інших показників використовуються додаток PowerTutor. Який вимірює споживання енергії на різних компонентах телефону, а також загальне використання пам'яті телефону. Він також дозволяє вимірювати енергоспоживання процесора конкретним застосуванням.

Для того щоб забезпечити справедливість тестів, для застосувань мовою Scala необхідно скористатись інструментом Proguard.

Proguard - інструмент, який "виявляє і видаляє класи, поля, методи і атрибути, що не використовуються. Він оптимізує байт-код і видаляє невживані інструкції, перейменовує класи, поля і методи з використанням короткої безглуздої назви." Це необхідний інструмент для розробки з Scala на Android, тому що включення цілої бібліотеки Scala збільшує час запуску програми і робить розмір набагато більшим.

Тестовий пристрій - HTC Evo, з версією Android 4.3, що має частоту процесора 1.2 ГГц та Meizu m2 note з версією Android 5.1, що має частоту процесора 1.3 ГГц. За допомогою цих тестових пристроїв ми можемо проаналізувати роботу застосувань на Dalvik та на ART.

Отже, на основі цих двох застосувань ми можемо порівняти всі ключові аспекти двох мов. За допомогою першого порівнюється поведінка структури коду, при написанні великого проекту. Другий дозволяє якісно оцінити роботу застосування на мобільному пристрої на основі вимірів використання пам’яті, обчислювальної здатності, тощо.

2.2 Результати тестування

2.2.1 Об’єм та структура коду

Для порівняння об’єму коду та складності структури пораховано кількість файлів у проекті, а також кількість рядків.

Отже, з малюнків 2.1 і 2.2 ми бачимо що кількість файлів у Scala трошки більше ніж у два рази меньша за Java, а також у півтора рази меньше кількість рядків. Необхідно також зазначити, що у проекті не змінювалась частина логіки і використовувався Java код, частка якого складає 60%.

В основному, різке зменьшення об’єму коду це результат використання case класів Scala, що дозволяють уникнути рутинних дій при оголошенні структури класів моделей і описати всю структуру інформації, що передається й отримується, в одному файлі.

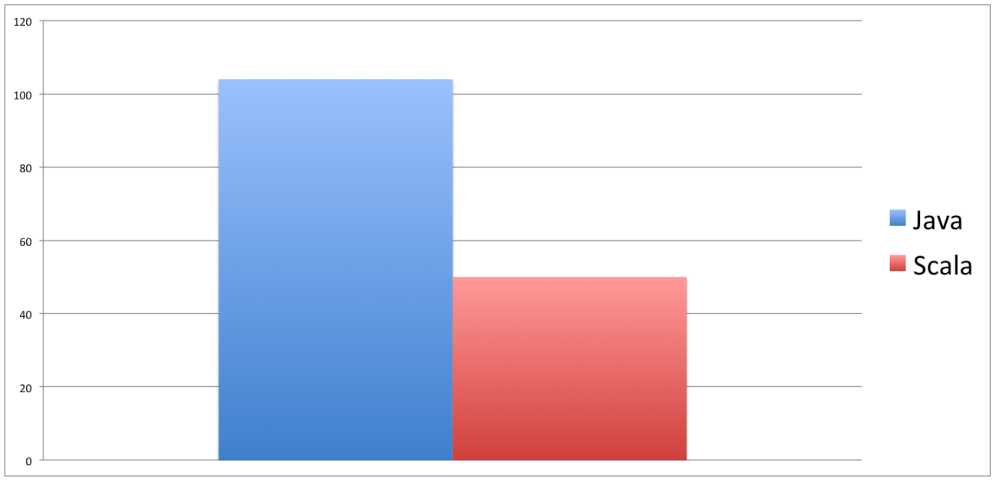

Рисунок 2.1 – Кількість файлів у проектах

Оголошення класу на Java займає у 58 разів більше, ніж на Scala.

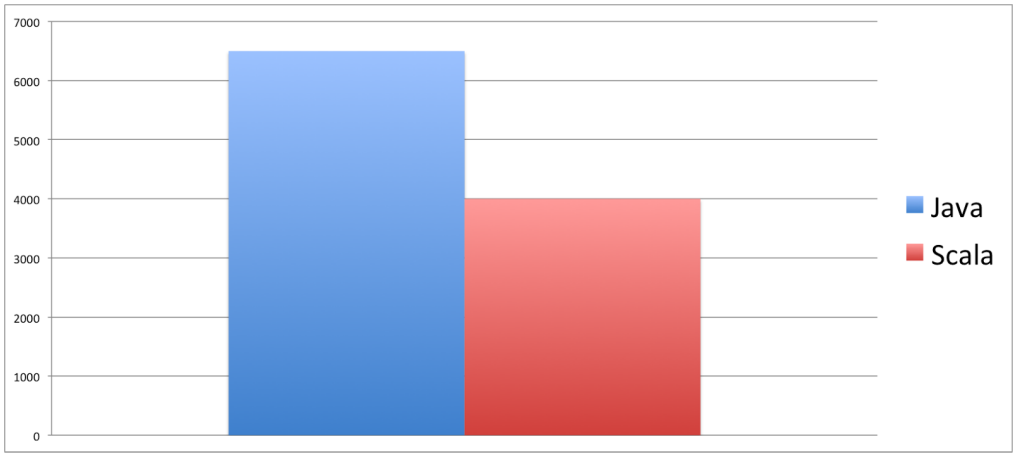

Рисунок 2.2 – Кількість рядків коду у проектах

Звісно, це не єдина причина. Також відчутний вплив монад при роботі з масивами, вони дозволяють зменшити кількість написаного коду в двічі, в тричі.

2.2.2 Продуктивність

З малюнка 2.3 бачимо що Java має стабільну швидкість написання коду, кожного тижня було зроблено як мінімум 1 завдання. Scala ж має трохи інші

показники. Так як до уваги взято спільні для обох мов завдання, ми бачимо що перші 3 тижні прогрес майже не рухався. Це результат того що в ці тижні створювалась DSL (domain specific language) для автоматизації рутинних, багатослівних дій, проводилась адаптація інструментів для роботи з Android SDK та сторонніми бібліотеками. Після цих робіт спостерігається різкий скачок, він пояснюється тим, що з використанням нової DSL зменшилась кількість рутинних дій, а також за рахунок уже написаної бізнес логіки на Java. Вже на шостий тиждень весь функціонал попередньої версії був переписаний на Scala.

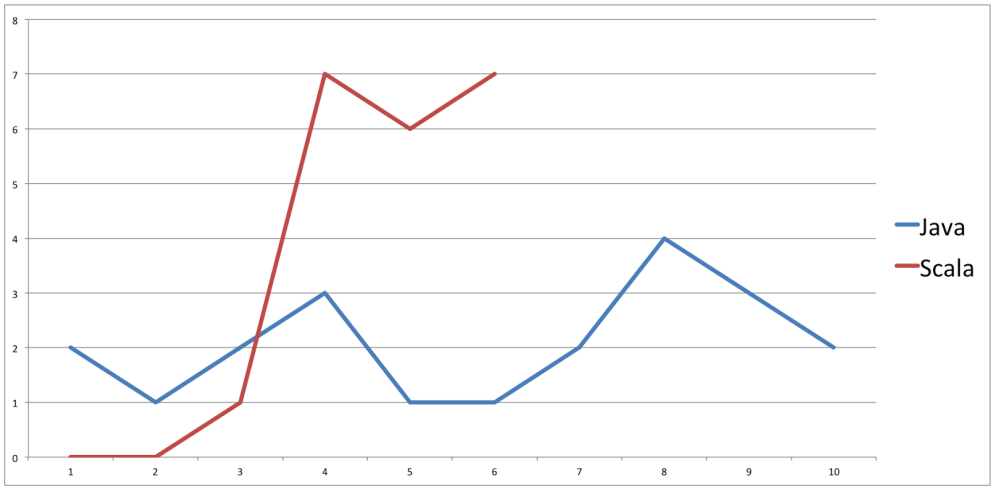

Рисунок 2.3 – Щотижнева кількість виконаних завдань при написанні проектів

З цього можемо зробити висновок про те що Scala потребує менше зусиль у рівних умовах написання нової фукнціональності. Також необхідно зазначити, що за час роботи над проектом, кількість помилок при тестуванні зменшилась у півтора рази. Покращилась надійність при роботі з мережею, адже RxJava надає гнучкий механізм обробки типових помилок, а також дозволяє відправляти декілька запитів, використовуючи попередню інформацію. Додавання RxJava вирішило проблему великої кількості обробників помилок для асинхронних запитів, що відкинуло потребу у значній кількості класів та полегшило структуру коду.

2.2.3 Розмір застосування, час запуску та встановлення

Для можливості використання Scala в Android застосування, необхідно підключити Scala plugin. Але він має досить суттєві розміри, вирішити цю проблему можна за допомогою використання ProGuard. Це відбувається шляхом видалення зайвих частин бібліотек, що не використовуються.

Таб. 2.1 – Таблиця розмірів застосувань

Мова	Розмір застосування проекту	Розмір застосування для вимірів
Java	4 106 KB	180 KB
Scala	7 367 KB	4041 KB
Java + Proguard	3 506 KB	180 KB
Scala + Proguard	4 028 KB	568 KB

У таблиці 2.1 показано, що використання Proguard для проекту значно зменьшує розміри застосування проекту для обох мов. У другому випадку, Proguard не впливає на розмір програми Java, у той же час значно зменшуючи розмір програми Scala. Однак, у порівнянні з Java, розмір програми Scala залишається на 400-500 KB більшим, хоча сам код має менший об’єм. При чому на застосуваннях середнього та великого розмірів ця різниця майже непомітна.

Час запуску для всіх застосувань залишився приблизно однаковим, з чого можемо зробити висновок що на цей параметр наявність Scala бібліотеки ніяк не впливає. Відповідно час встановлення для Scala без ProGuard збільшився пропорційно розмірам.

2.2.4 Час виконання

Зрозуміло, що ми не зможемо оцінити час виконання проекту, адже він має розгалуджену структуру, а також тісно взаємодіє з мережею, а отже його час виконання повністю залежить від якості інтернет з'єднання в даний момент часу.

Додаток із двома контрольними задачами відстежує час виконання кожної ітерації алгоритмів.

На Dalvik VM, час виконання N-Body проблеми описаній за допомогою Java виконується значно швидше, ніж версія написана на Scala. (Рис. 2.4)

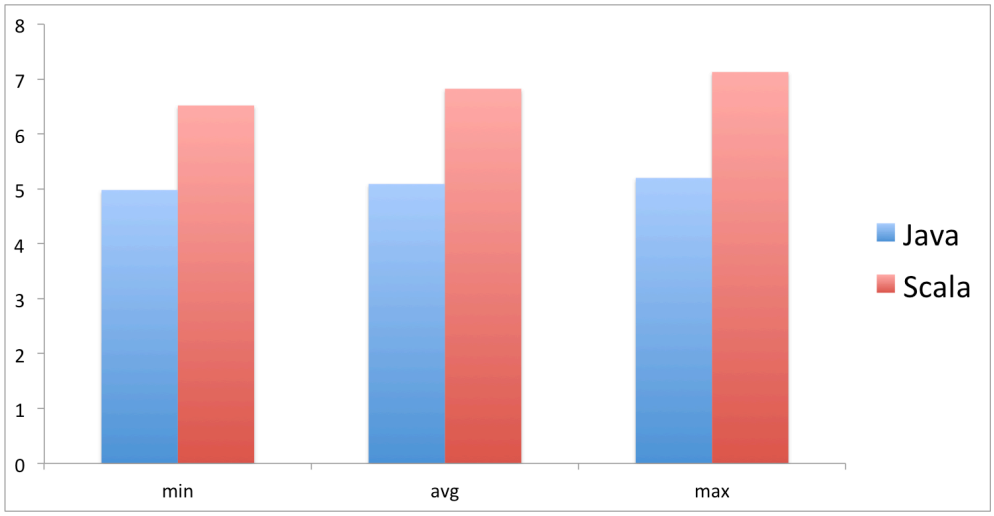

Рисунок 2.4 – Порівняння часу виконання N-Body задачі Dalvik

На ART, ми бачимо таку ж ситуацію, хоча й час виконання зменшився, це відбулось за рахунок середовища. (Рис. 2.5)

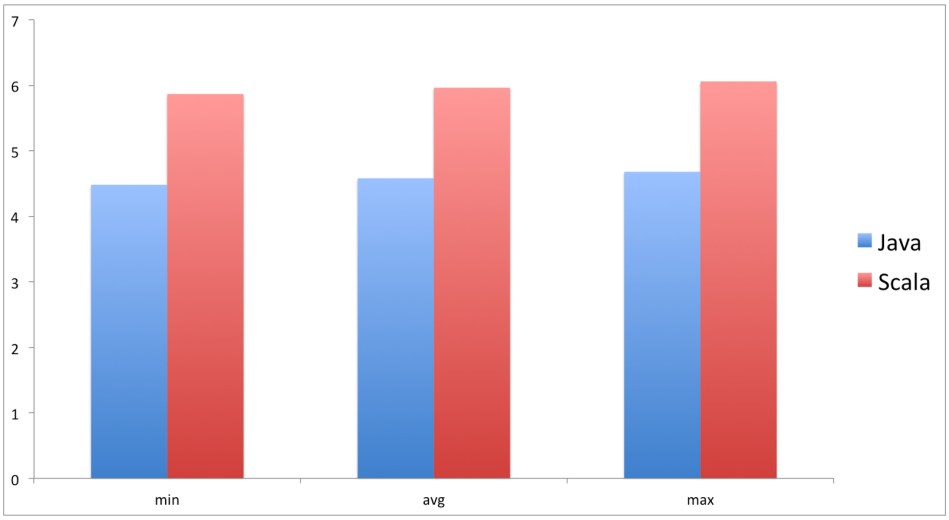

Рисунок 2.5 – Порівняння часу виконання N-Body задачі ART

Контрольна задача роботи з бінарними деревами працює на Scala трохи швидше ніж на Java. Так як ця задача націлена на тестування роботи з пам'яттю, а також робот у збирача сміття, можемо зробити висновок що Scala код краще оптимізований для роботи з пам'яттю, але в той же час показує трохи гірший результат, аніж Java для складних обчислень.

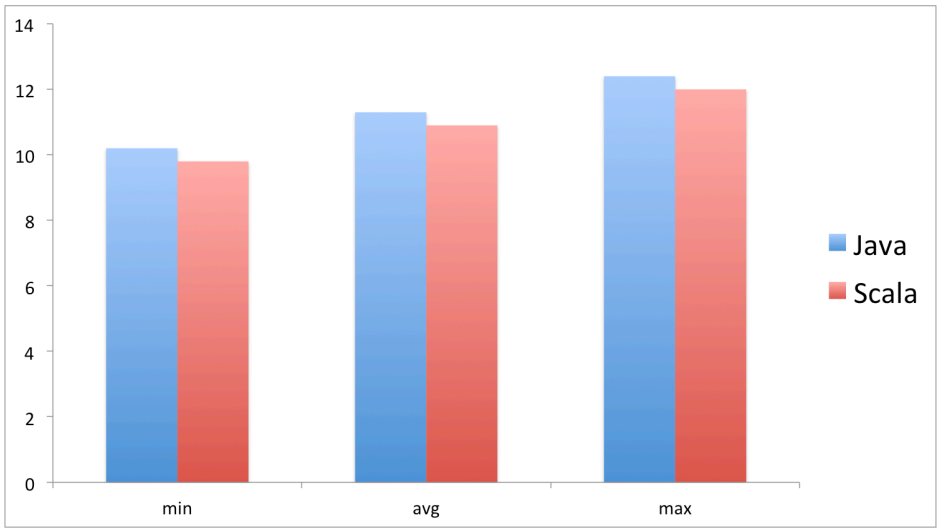

Рисунок 2.6 – Порівняння часу виконання задачі на бінарних деревах Dalvik

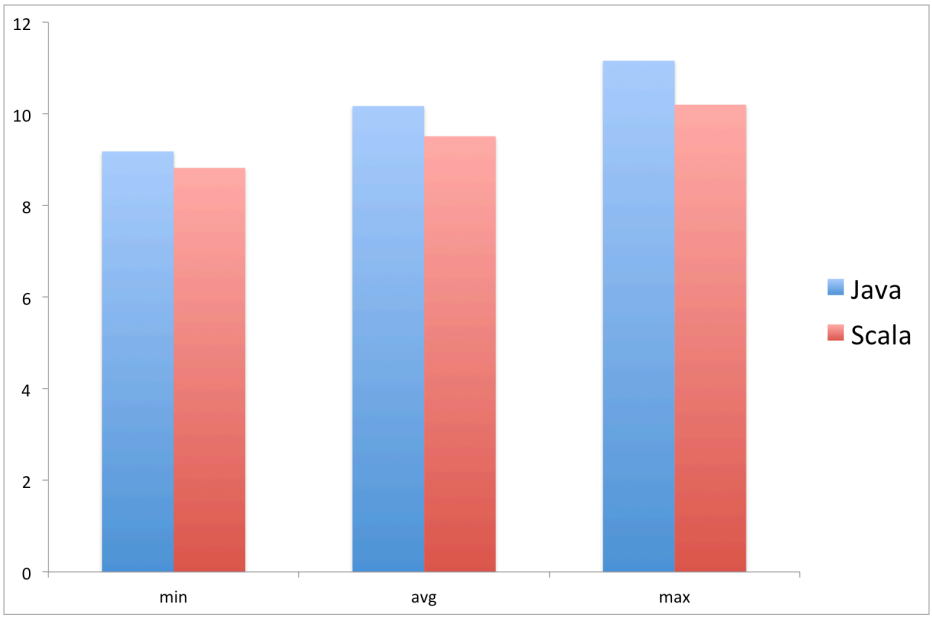

Рисунок 2.7 – Порівняння часу виконання задачі на бінарних деревах ART

На ART середовищі ми бачимо більшу різницю на користь Scala.

Важкі обчислення, такі як N-body проблем дуже рідко зустрічаються у реальному житті, адже завдання що потребують більшої обчислювальної потужності переносяться на серверну частину. Тому оптимізація пам'яті та зручність роботи з мережею є більш вагомими перевагами.

2.2.5 Споживання енергії

Отже, для контрольних задач, виміряно споживання акамулятора, що виключає роботу екрану та мережі, адже алгоритми їх не використовують.

При вимірюванні витрати акамуляторана версії Java в N-body задачі, споживання має в середньому 1017mW, і загальний час виконання для 10 серій 49 секунд на Dalvik машині. На ART - 904mW і час виконання 41 секунду.

Версія Scala має середнє енергоспоживання 1004mW, і загальний час виконання 66.5 секунд. На Dalvik машині. На ART - 894mW і час виконання 60 секунд.

Як бачимо, результати дуже схожі. Версія Scala потребує трохи менше енергії, але за рахунок того що виконується повільніше - споживає в результаті трохи більше.

Для тесту за допомогою бінарних дерев виявлено, що версія Java має середнє споживання енергії на Dalvik машині 785mW, загальний час виконання для прогонів даного тесту - 118 секунд. Версія Scala має середнє споживання 889mW, із загальним часом виконання 113.3 секунд. На ART Java - 736mW, час виконання 113 секунд, Scala - 856mW, час виконання 104 секунди.

Порівнюючи результати для двох мов, можемо бачити, що версія Scala для контрольної задачі на бінарних деревах має більш високе енергоспоживання. Однак, не дивлячись на те, що час виконання версії Scala меньший ніж Java, загальне споживання енергії кожного пробігу значно енергоємнісний, через високу середню вживану потужність.

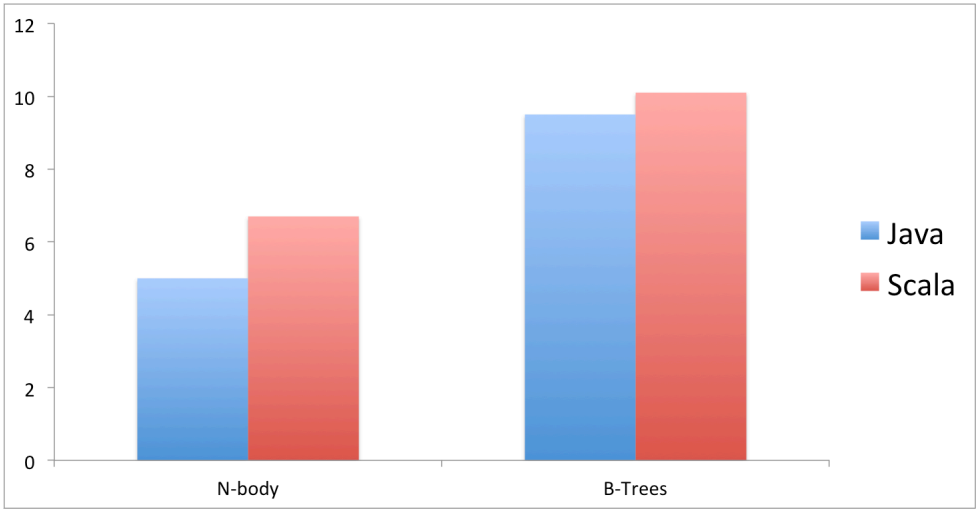

Рисунок 2.8 – Порівняння споживання енергії застосуванням Java та Scala у Джоулях Dalvik

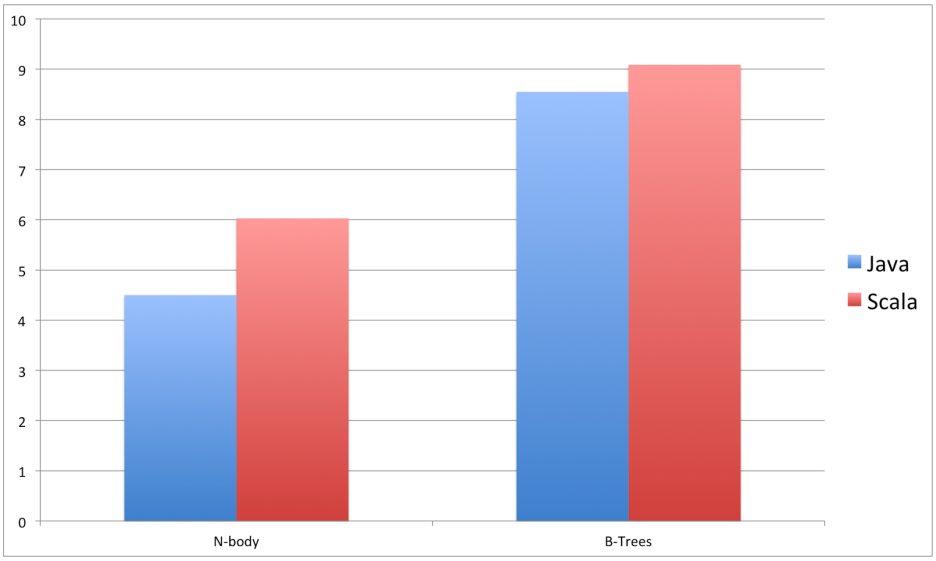

Рисунок 2.9 – Порівняння споживання енергії застосуванням Java та Scala у Джоулях ART

2.2.6 Використання пам’яті

На графіку (рис 2.9 та 2.10), значення між 1-м і 6-м читанням пам’яті відповідають виконанням N-Body проблеми, значення між 7-м і 18-м читанням відповідають контрольному завданню на бінарних деревах.

За допомогою цих даних, можемо бачити, що система має набагато більше вільної пам’яті під час виконання коду Scala, аніж Java. Однією з причин цього є поведінка збирача сміття: об’єкти в програмі Scala мають менші розміри і залишаються в системі значно менше часу, аніж Java об’єкти, а отже збирач сміття працює більш продуктивно.

Також, ми можемо помітити подібність поведінки застосувань Java і Scala під час виконання контрольних завдань, з низьким використанням пам'яті - N-Body проблема і постійного використання пам’яті під час тесту на бінарних деревах.

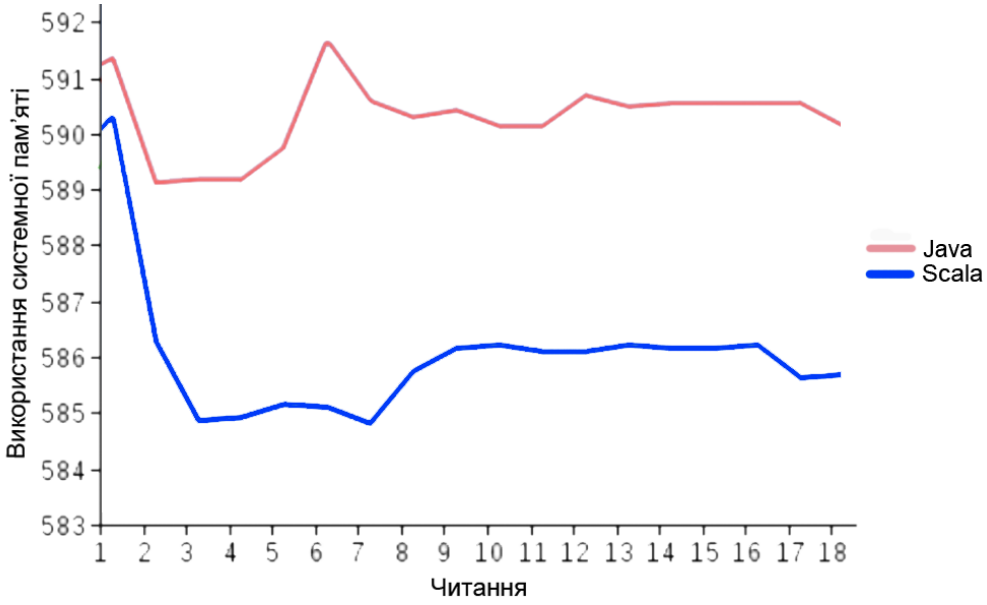

Рисунок 2.10 – Порівняння кількості використаної пам’яті застосуванням Java та Scala Dalvik

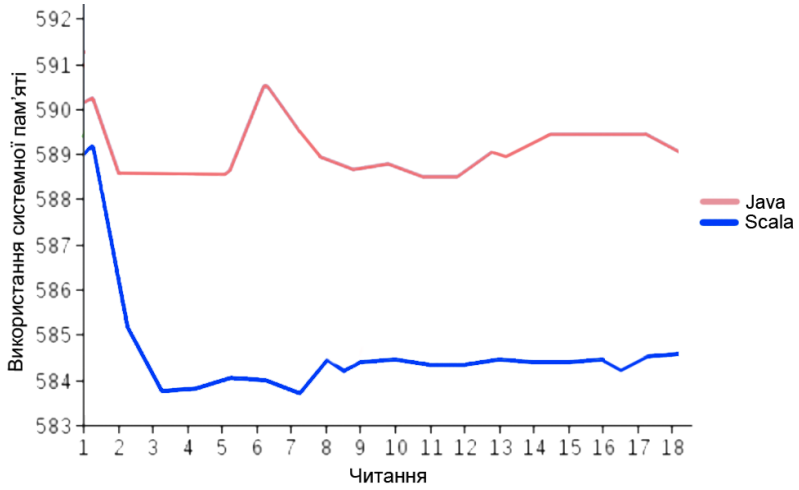

Рисунок 2.11 – Порівняння кількості використаної пам’яті застосуванням Java та Scala ART

2.3 Висновок

Отже, на основі контрольних тестів, що порівнювали мови за такими критеріями:

- об’єм коду;
- трудозатратність;
- час виконання;

- споживання енергії;
- використання оперативної пам’яті;
- розмір застосування;
- час запуску застосування.

Можемо зробити висновок що Scala більш лаконічна мова. За рахунок нових структур об’єм коду зменшується в середньому в 2-3 рази, а також спрощує структуру ліквідовуючи засмічення проекту великою кількістю рутинних службових класів та методів. Scala потребує менших зусиль при написанні коду, час на розробку також зменшується в 1.5-2 рази. Використання реактивної парадигми RxJava оптимізує та спрощує оброку запитів та помилок.

Scala показує себе гірше, аніж Java в задачах де необхідна значна процесорна потужність, але краще оптимізована для роботи в умовах обмеження пам’яті. Зважаючи на те, що задачі пов’язані зі складними обчисленнями зустрічаються на клієнтській стороні досить рідко, адже подібні задачі виносяться на більш стабільну та потужну серверну частину, оптимізація пам’яті та роботи з мережею є більш важливими перевагами для мобільних застосувань. Споживання енергії застосуваннями не має значних відмінностей у роботі двох мов, хоча Java використовує трохи меньше енергії.

Отже, більш раціональним вибором при пошуку аналогів Java для розробки стандартних мобільних застосувань без складних обчислень є Scala, яка поєднує в собі переваги сучасної мови з можливістю використання старої кодової бази написаної на Java, а також наслідує всі перевірені часом плюси Java.

3 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка засобів розробки програмних продуктів для Android платформи.

Нижче наведено аналіз різних варіантів реалізації Android застосувань з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики застосування, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. Як прямі, так і побічні витрати розподіляються по продуктам та послугам у залежності від потрібних на кожному етапі виробництва обсягів ресурсів. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

- визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

- для кожної функції визначаються повні річні витрати й кількість робочих часів.
- для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.
- після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

3.1 Постановка задачі техніко-економічного аналізу

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу засобів розробки програмних продуктів для Android платформи.

Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

- програмний продукт повинен функціонувати на мобільних пристроях із стандартним набором компонент;
- забезпечувати високу швидкість обробки великих об’ємів даних у реальному часі;
- забезпечувати зручність і простоту взаємодії з користувачем;
- передбачати мінімальні витрати на впровадження програмного продукту.

3.1.1 Обґрунтування функцій програмного продукту

У даному розділі проводиться оцінка засобів розробки програмних продуктів для Android платформи.

Нижче наведено аналіз різних варіантів реалізації Android застосувань з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики застосування, що впливають на продуктивність роботи і на

його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. Як прямі, так і побічні витрати розподіляються по продуктам та послугам у залежності від потрібних на кожному етапі виробництва обсягів ресурсів. Виконані на цих етапах дії у контексті метода ФВА називаються функціями.

Мета ФВА полягає у забезпеченні правильного розподілу ресурсів, виділених на виробництво продукції або надання послуг, на прямі та непрямі витрати. У даному випадку – аналізу функцій програмного продукту й виявлення усіх витрат на реалізацію цих функцій.

Фактично цей метод працює за таким алгоритмом:

– визначається послідовність функцій, необхідних для виробництва продукту. Спочатку – всі можливі, потім вони розподіляються по двом групам: ті, що впливають на вартість продукту і ті, що не впливають. На цьому ж етапі оптимізується сама послідовність скороченням кроків, що не впливають на цінність і відповідно витрат.

– для кожної функції визначаються повні річні витрати й кількість робочих часів.

– для кожної функції наоснові оцінок попереднього пункту визначається кількісна характеристика джерел витрат.

– після того, як для кожної функції будуть визначені їх джерела витрат, проводиться кінцевий розрахунок витрат на виробництво продукту.

3.2 Постановка задачі техніко-економічного аналізу

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу засобів розробки програмних продуктів для Android платформи.

Відповідно цьому варто обирати і систему показників якості програмного продукту.

Технічні вимоги до продукту наступні:

- програмний продукт повинен функціонувати на мобільних пристроях із стандартним набором компонент;
- забезпечувати високу швидкість обробки великих об'ємів даних у реальному часі;
- забезпечувати зручність і простоту взаємодії з користувачем;
- передбачати мінімальні витрати на впровадження програмного продукту.

3.2.1 Обґрунтування функцій програмного продукту

Головна функція F_0– розробка абстрактного програмного продукту, який отримує, обробляє користувацькі дані, використовує анімацію, тощо. Виходячи з конкретної мети, можна виділити наступні основні функції ПП:

F_1 – вибір мови програмування;

F_2 – вибір способу екранної розмітки;

F_3 – вибір парадигми програмування.

Кожна з основних функцій може мати декілька варіантів реалізації.

Функція F_1:

а) написання екранної розмітки мовою XML;

б) написання екранної розмітки за допомогою DSL;

Функція F_2:

а) використання функціональної парадигми;

б) використання імперативної парадигми.

Функція F_3:

а) мова програмування Scala;

б) мова програмування Java;

3.2.2 Варіанти реалізації основних функцій

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 3.1). На основі цієї карти побудовано позитивно-негативну матрицю варіантів основних функцій (таблиця 3.1).

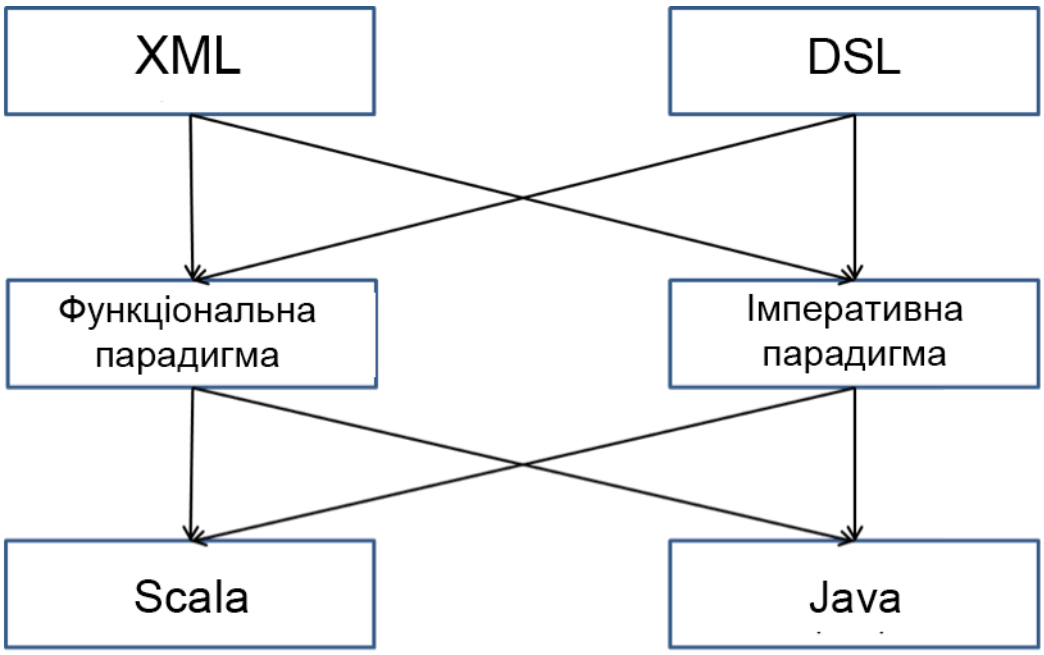

Рисунок 3.1 – Морфологічна карта

Морфологічна карта відображує всі можливі комбінації варіантів реалізації функцій, які складають повну множину варіантів ПП.

Таблиця 3.1 – Позитивно-негативна матриця

Основні функції	Варіанти реалізації	Переваги	Недоліки
F1	А	Велика кодова база, користувацька підтримка	Відсутня модульність, об’ємність коду
	Б	Можлива модульність та краще повторне використання коду	Немає прекомпільованого показу коду, більше часу для написання
F2	А	Більш стабільна та швидка	Велика вірогідність помили, складніша архітектура
	Б	Клієнтська підтримка, кодова база	Більший поріг входження
F3	А	Займає менше часу при написанні коду, може використовувати код Java	Вимагає більшої кількості пам’яті, незвична для більшості програмістів
	Б	Кодова база, підтримка IDE	Займає більше часу при написанні коду, об’єм коду

На основі аналізу позитивно-негативної матриці робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому, що вони не відповідають поставленим перед програмним продуктом задачам.

Функція *F1:*

Оскільки фактор швидкості та надійності при розробці мобільних застосувань вважається головним, а б) жодному з них не відповідає, він може бути відкинутим.

Функція *F2:*

Оскільки 2 варіанта мають суперечливі характеристики, вони є гідними до розгляду.

Функція *F3:*

Оскільки написання програмного забезпечення за допомогою Scala не забороняє використання Java та старої кодової бази, метод б) ми також можемо відкинути.

Таким чином, будемо розглядати такі варіанти реалізації ПП:

F1a – F2a – F3a

F1a – F2б – F3a

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

3.3 Обґрунтування системи параметрів ПП

3.3.1 Опис параметрів

На підставі даних про основні функції, що повинен реалізувати програмний продукт, вимог до нього, визначаються основні параметри виробу, що будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

X1 – швидкодія мови програмування;

X2 – об’єм пам’яті для збереження даних;

X3 – час обробки даних;

X4 – потенційний об’єм програмного коду.

X1: Відображає швидкодію операцій залежно від обраної мови програмування.

X2: Відображає об’єм пам’яті в оперативній пам’яті персонального комп’ютера, необхідний для збереження та обробки даних під час виконання програми.

X3: Відображає час, який витрачається на дії.

*X4:*Показує розмір програмного коду який необхідно створити безпосередньо розробнику.

3.3.2 Кількісна оцінка параметрів

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у табл. 3.2.

Таблиця 3.2 – Основні параметри ПП

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія мови програмування	X1	Оп/мс	2000	12000	19000
Об’єм пам’яті для збереження даних	X2	Мб	32	16	8
Час обробки даних алгоритмом	X3	Мс	800	420	60
Потенційний об’єм програмного коду	X4	кількість строк коду	2000	1500	1000

За даними таблиці 3.2 будуються графічні характеристики параметрів – рис. 3.2 – рис. 3.5.

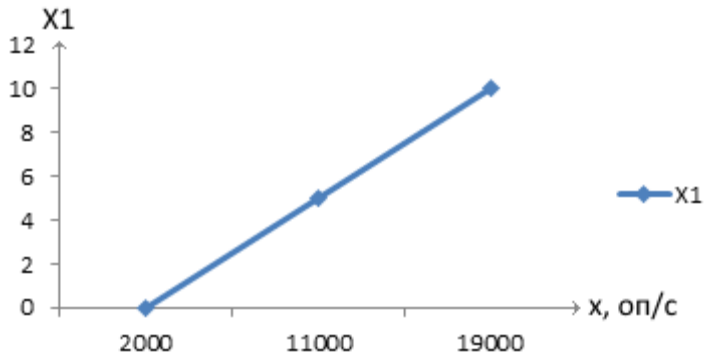

Рисунок 3.2 – X1, швидкодія мови програмування

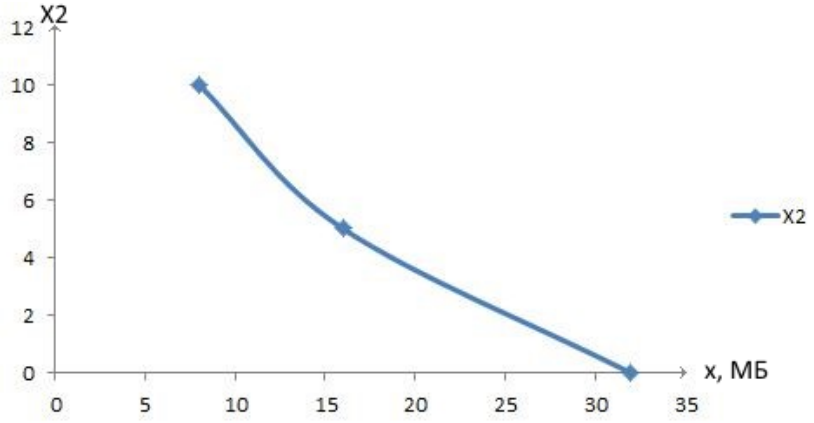

Рисунок 3.3 – X2, об’єм пам’яті для збереження даних

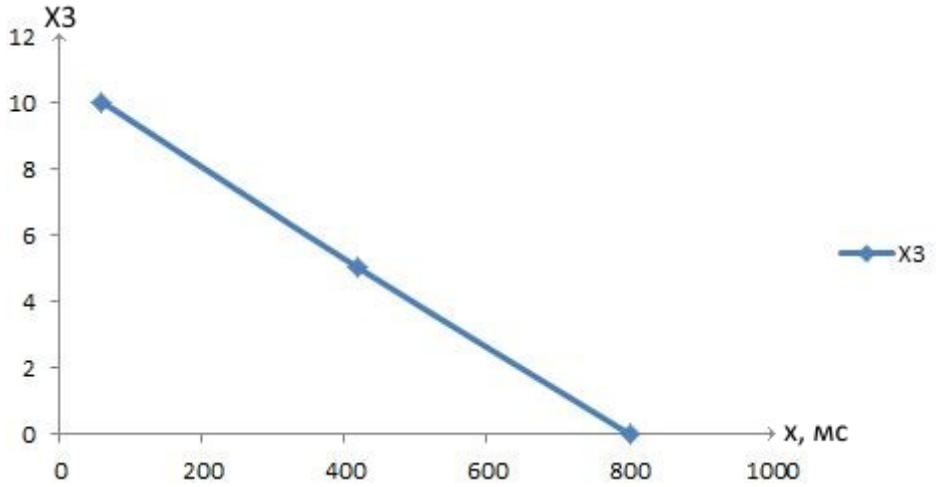

Рисунок 3.4 – X3, час обробки даних алгоритмом

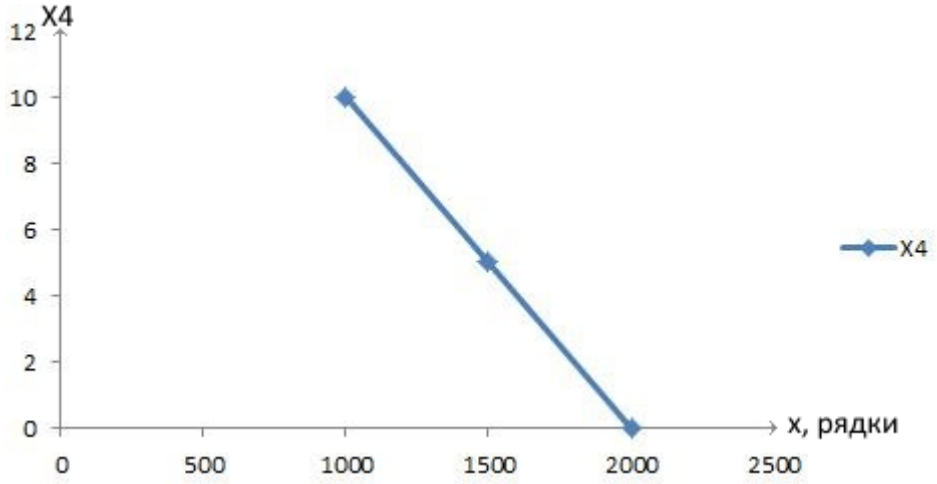

Рисунок 3.5 – X4, потенційний об’єм програмного коду

3.3.3 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який дає найбільш точні результати при знаходженні параметрів моделей адаптивного прогнозування і обчислення прогнозних значень.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

- визначення рівня значимості параметра шляхом присвоєння різних рангів;
- перевірку придатності експертних оцінок для подальшого використання;
- визначення оцінки попарного пріоритету параметрів;
- обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 3.3.

Таблиця 3.3 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангі в R_i	Відхилення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	4	3	4	4	4	4	4	27	0.75	0.56
X2	Об'єм пам'яті для збереження даних	Мб	4	4	4	3	4	3	3	25	-1.25	1.56
X3	Час обробки даних алгоритмом	Мс	2	2	1	2	1	2	2	12	-14.25	203.06
X4	Потенційний об'єм програмного коду	Кількість рядків коду	5	6	6	6	6	6	6	41	14.75	217.56
	Разом		15	15	15	15	15	15	15	105	0	420.75

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 105,$$

де N – число експертів, n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 26.25$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 420.75$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 420.75}{7^2(5^3 - 5)} = 1{,}03 > W_k = 0.67$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0.67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 3.4.

Таблиця 3.4 – Попарне порівняння параметрів

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	=	>	=	<	=	<	<	<	0.5
X1 і X3	<	<	<	<	<	<	<	<	0.5
X1 і X4	>	>	>	>	>	>	>	>	1.5
X2 і X3	<	<	<	<	<	<	<	<	0.5
X2 і X4	>	>	>	>	>	>	>	>	1.5
X3 і X4	>	>	>	>	>	>	>	>	1.5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij}визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}$$

З отриманих числових оцінок переваги складемо матрицю $A = \| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{ві}$ за наступними формулами:

$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i}$,де $b_i = \sum_{i=1}^{N} a_{ij}$.

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%).На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}$,де $b_i' = \sum_{i=1}^{N} a_{ij} b_j$.

Як видно з таблиці 3.5, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 3.5 – Розрахунок вагомості параметрів

Параметри x_i	Параметри x_j				Перша ітер.		Друга ітер.		Третя ітер	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$	b_i^2	$K_{\text{ві}}^2$
X1	1.0	0.5	0.5	1.5	3.5	0.219	22.25	0.216	100	0.215
X2	1.5	1.0	0.5	1.5	4.5	0.281	27.25	0.282	124.25	0.283
X3	1.5	1.5	1.0	1.5	5.5	0.344	34.25	0.347	156	0.348
X4	0.5	0.5	0.5	1.0	2.5	0.156	14.25	0.155	64.75	0.154
Всього:					16	1	98	1	445	1

3.3 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2*(об’єм пам’яті для збереження даних) *X1* (швидкодія мови програмування) та *X3* відповідають технічним вимогам умов функціонування даного ПП.1

Абсолютне значення параметра *X4 (*потенційний об’єм програмного коду) обрано не найгіршим (не максимальним), тобто це значення відповідає або варіанту а) 1800 або варіанту б) 1200.

Таблиця 3.6 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1(X1)	А	12000	3.6	0.215	0.674
F2(X3)	А, Б	800	2.4	0.348	0.835
F2(X4)	А	1800	2	0.154	0.308
	Б	1200	8	0.154	1.232
F3(X2)	Б	16	3.4	0.283	0.962

За даними з таблиці 3.6 за формулою

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}],$$

визначаємо рівень якості кожного з варіантів:

$K_{K1} = 0.674 + 0.835 + 0.308 + 0.962 = 2.628$

$K_{K2} = 0.674 + 0.835 + 1.232 + 0.962 = 3.723$

Як видно з розрахунків, кращим є другий варіант, для якого коефіцієнт технічного рівня має найбільше значення.

3.4 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

При цьому варіант 3 має додаткове завдання:

3 Реалізація методів аналізу;

4 А варіант 4 має інше додаткове завдання:

5 Обробка інтерфейсу готових бібліотек.

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 2.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань. Загальна трудомісткість обчислюється як

$$T_O = T_P \cdot K_П \cdot K_{СК} \cdot K_M \cdot K_{СТ} \cdot K_{СТ.М}, \qquad (5.1)$$

де T_P – трудомісткість розробки ПП;$K_П$ – поправочний коефіцієнт;$K_{СК}$ – коефіцієнт на складність вхідної інформації; K_M – коефіцієнт рівня мови програмування;$K_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;$K_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: T_P =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $K_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх завдань рівний 1: $K_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за

допомогою коефіцієнта $К_{СТ}$ = 0.8. Тоді, за формулою 5.1, загальна трудомісткість програмування першого завдання дорівнює:

$Т_1$ = 90·1.7·0.8 = 122.4 людино-днів.

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм другої групи складності, степінь новизни Б), тобто $Т_Р$ =27 людино-днів, $К_П$ =0.9,$К_{СК}$ = 1,$К_{СТ}$ =0.8:

$Т_2$ = 27 · 0.9 · 0.8 = 19.44 людино-днів.

Для третього завдання (використовується алгоритм другої групи складності, ступінь новизни Г з використанням перемінної інформації):

Тр = 12 людино-днів; Кп = 0.72; Кст = 0.8;

То = 12 • 0.72 • 0,8 = 6.91.

Для четвертого завдання (використовується алгоритм третьої групи складності, ступінь новизни Г):

Тр = 8 людино-днів; Кп = 0.6; Кст = 1;

То = 8 • 0.6 • 1 = 4.8.

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$Т_I$ = (122.4 + 19.44 + 6.91) · 8 = 1190 людино-годин;

$Т_{II}$ = (122.4 + 19.44 + 4.8) · 8 = 1173.12 людино-годин;

Найбільш високу трудомісткість має варіант I.

В розробці беруть участь два програмісти з окладом 6000 грн., один фінансовий аналітик з окладом 9000 грн. Визначимо зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m \cdot t} грн.,$$

де М – місячний оклад працівників; T_m – кількість робочих днів тиждень; t – кількість робочих годин в день.

$$С_ч = \frac{6000 + 6000 + 9000}{3 \cdot 21 \cdot 8} = 41.67 \text{ грн.}$$

Тоді, розрахуємо заробітну плату за формулою

$$С_{ЗП} = С_ч \cdot Т_i \cdot К_Д,$$

де $С_ч$– величина погодинної оплати праці програміста;$Т_i$ – трудомісткість відповідного завдання; $К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

I. $С_{ЗП} = 41.67 \cdot 1190 \cdot 1.2 = 59504.76$ грн.

II. $С_{ЗП} = 41.67 \cdot 1173.12 \cdot 1.2 = 58660.69$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

I. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 59504.76 \cdot 0.22 = 13091.05$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 58660.69 \cdot 0.22 = 12905.35$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 6000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 6000 \cdot 0.2 = 14400$ грн.

З урахуванням додаткової заробітної плати:

$С_{ЗП} = С_Г \cdot (1 + К_З) = 14400 \cdot (1 + 0.2) = 17280$ грн.

Відрахування на єдиний соціальний внесок:

$С_{ВІД} = С_{ЗП} \cdot 0.22 = 17280 \cdot 0.22 = 3801.6$ грн.

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 8000 грн.

$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 8000 = 2300$ грн.,

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача; $К_А$– річна норма амортизації; $Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 8000 \cdot 0.05 = 460$ грн.,

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$Т_{ЕФ} = (Д_К - Д_В - Д_С - Д_Р) \cdot t_З \cdot К_В = (365 - 104 - 8 - 16) \cdot 8 \cdot 0.9 = 1706.4$ годин,

де $Д_К$ – календарна кількість днів у році; $Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів; $Д_Р$ – кількість днів планових ремонтів устаткування; t –кількість робочих годин в день; $К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1706.4 \cdot 0.156 \cdot 0.2 \cdot 2.0218 = 107.64$ грн.,

де N_C – середньо-споживча потужність приладу; $K_З$– коефіцієнтом зайнятості приладу; $Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 8000 \cdot 0.67 = 5360 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н$$

$$С_{ЕКС} = 17280 + 3801.6 + 2300 + 460 + 107.64 + 5360 = 29309.24 \text{ грн.}$$

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 29309.24 / 1706.4 = 17.18 \text{ грн/час.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$С_М = С_{М\text{-}Г} \cdot Т$$

I. $С_М = 17.18 \cdot 1190 = 20444.2$ грн.;

II. $С_М = 17.18 \cdot 1173.12 = 20154.2$ грн.;

Накладні витрати складають 67% від заробітної плати:

$$С_Н = С_{ЗП} \cdot 0.67$$

I. $С_Н = 59504.76 \cdot 0.67 = 39868.19$ грн.;

II. $С_Н = 58660.69 \cdot 0.67 = 39302.66$ грн.;

Отже, вартість розробки ПП за варіантами становить:

$$С_{ПП} = С_{ЗП} + С_{ВІД} + С_М + С_Н$$

I. $С_{ПП} = 59504.76 + 13091.05 + 20444.2 + 39868.19 = 132908.2$ грн.;

II. $С_{ПП}$ = 58660.69 + 12905.35 + 20154.2 + 39302.66 = 131022.9 грн.;

3.5 Вибір кращого варіанта ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$К_{ТЕРj} = К_{Кj} / С_{Фj}$,

$К_{ТЕР1}$ = 2.628 / 132908.2 = $0.19 \cdot 10^{-4}$;

$К_{ТЕР2}$ = 3.723 / 131022.9= $0.27 \cdot 10^{-4}$;

Як бачимо, найбільш ефективним є другий варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 0.27 \cdot 10^{-4}$.

3.4 Висновок

В даному розділі проведено повний функціонально-вартісний аналіз ПП, який було розроблено в рамках дипломного проекту. Процес аналізу можна умовно розділити на дві частини.

В першій з них проведено дослідження ПП з технічної точки зору: було визначено основні функції ПП та сформовано множину варіантів їх реалізації; на основі обчислених значень параметрів, а також експертних оцінок їх важливості було обчислено коефіцієнт технічного рівня, який і дав змогу визначити оптимальну з технічної точки зору альтернативу реалізації функцій ПП.

Другу частину ФВА присвячено вибору із альтернативних варіантів реалізації найбільш економічно обґрунтованого. Порівняння запропонованих варіантів реалізації в рамках даної частини виконувалось за коефіцієнтом ефективності, для обчислення якого були обчислені такі допоміжні параметри, як трудомісткість, витрати на заробітну плату, накладні витрати.

Після виконання функціонально-вартісного аналізу програмного комплексу що розроблюється, можна зробити висновок, що з альтернатив, що залишились після першого відбору двох варіантів виконання програмного комплексу оптимальним є перший варіант реалізації програмного продукту. У нього виявився найкращий показник техніко-економічного рівня якості $К_{ТЕР} = 0.27 \cdot 10^{-4}$.

Цей варіант реалізації програмного продукту має такі параметри:

- мова розмітки – XML;
- функціональна парадигма програмування;
- мова програмування – Scala.

Тобто в нашому випадку спосіб з використанням Scala та функціональної парадигми – виявився більш вигідним з економічної точки зору.

ВИСНОВКИ

Метою роботи було підвищення ефективності створення Android застосувань.

Для досягнення результатів було обрано критерії порівняння властивостей фукнціональної парадигми при розробці мобільних застосувань. Для розширення Java було обрано фреймворки та бібліотеки: Functional Java, Retrolambda, RxJava. А також мови JVM: Clojure, Groovy, Scala. Під час їхнього порівняння виявилось що Functional Java - не підходить для розширення, адже потребує багато роботи зі старим кодом для імплементації в проект, а Retrolambda - нестабільно працює на Android. RxJava - значно розширює функціональність Java, а також стабільно працює на Android девайсах, але без використання Retrolambda збільшує об'єм коду в 2-3 рази. Серед мов було відкинуто Clojure, як Lisp подібну мову що не пристосовані для роботи з імперативним кодом Android SDK. Використання Groovy, що підтримує динамічну типізацію, збільшує час компіляції та вірогідність помилки. Scala зберігає всі переваги Java та ліквідовує недоліки.

Для необхідності порівняння Scala та Java було обрано такі критерії: об'єм коду, продуктивність розробки, час виконання, споживання енергії, використання оперативної пам'яті, розмір застосування, час запуску застосування.

Тестування проводилось в середовищі виконання Dalvik VM та ART, і показало наступні результати:

У середньому Scala на задачі зі складними обчисленнями час виконання на Dalvik на 25% більший ніж Java, на ART - на 23%.

Перевірка роботи з пам'ятю показала для Scala результат у обох випадках кращий: Dalvik на 3,7%, ART - 7%.

Під час перевірки енергоспоживання було виявлено що на Dalvik та ART результати майже однакові.

На складних обчисленнях Scala споживає на 25% більше енергії, на задачах із високим споживанням пам’яті на 5% більше.

Тести показали що Scala споживає в середньому в два рази менше пам’яті.

Отримані результати доводять доцільність використання Scala у поєднанні з RXJava для підвищення ефективності створення Android застосувань, у великих проектах загального призначення, для яких критичний час розробки, робота з мережею та мінімізація використання пам’яті. У проектах, для яких критичний розмір, використання енергії, час виконання та проведення складних обчислень - варто використовувати Java.

Підсумовуючи, слід сказати, що мета поставленої задачі досягнута і вирішена в повному обсязі.
Отриманий результат дозволить створювати застосування швидше та якісніше, краще обирати мову під цілі.

ПЕРЕЛІК ПОСИЛАНЬ

1. Smartphone OS Market Share. – Режим доступа: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. − Дата доступа: 15.03.2016.
2. Bornstein Dan. "Presentation of Dalvik VM Internals". – Atlanta / Big Nerd Ranch, 2010. - С. 22.
3. Android stack. – Режим доступа: https://github.com/devacademy/android-bootcamp/blob/master/modules/stack.md. − Дата доступа: 15.03.2016.
4. Android NDK. – Режим доступа: http://developer.android.com/intl/ru/tools/sdk/ndk/index.html. − Дата доступа: 15.03.2016.
5. Е. А. Роганов Основы информатики и программирования: Учебное пособие / Е. А. Роганов. — М.: МГИУ, 2001. — Вузол VI.
6. Лавріщева К. М. Програмна інженерія / Лавріщева К. М. — К.: Академперіодика, 2008.- 319 с.
7. R.W. Floyd. The Paradigms of Programming Communications of the ACM / R.W. Floyd. - Atlanta : Big Nerd Ranch, 1979. - 455 p.
8. Вольфенгаген В. Э. Конструкции языков программирования. Приемы описания. / Вольфенгаген В. Э. — М: АО «Центр ЮрИнфоР», 2001. — 276 с ISBN 5-89158-079-9.
9. А. Філд, П. Харрісон Функціональне програмування. / А. Філд, П. Харрісон. – К: Москва "Мир", 1993. – 55 с.
10. Getting Started with XL Fortran. – Режим доступа: http://www-01.ibm.com/support/docview.wss?uid=swg27045455&aid=1. - Дата доступа: 15.03.2016.
11. Tail Call Optimization. – Режим доступа: http://c2.com/cgi?TailCallOptimization. − Дата доступа: 15.03.2016.

12. Сутула О.В. FUNCTIONAL REACTIVE PARADIGM ADVANTAGES FOR ANDROID DEVELOPMENT / Сутула О.В. // International Scientific Journal. – 2015. – №9. – С. 59-61.
13. Сутула О.В Порівняння методів розробки для мобільної платформи Android. / Сутула О.В. // Системний аналіз та інформаційні технології : «САІТ-2016», 30 травня –2 червня 2016, Київ, Україна : матеріали. – К. : НТУУ «КПІ», 2016. – С. 113
14. V. Pankratius, F. Schmidt, and G. Garreton Combining functional and imperative programming for multi-core software: An empirical study evaluating scala and java. / V. Pankratius, F. Schmidt, and G. Garreton // In Software Engineering (ICSE), 2012 34th International Conference, pages 123 –133, june 2012.
15. A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer New scala() instanceof java: A comparison of the memory behaviour of java and scala programs. / A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer // International Symposium on Memory Management, ISMM ’12, pages 97–108, New York, NY, USA, 2012. ACM.
16. Bill Phillips Android Programming: The Big Nerd Ranch Guide (2nd Edition) / Bill Phillips. - Atlanta : Big Nerd Ranch, 2015. - 600 p.
17. Greg Michaelson An introduction to functional programming through lambda calculus / Greg Michaelson // - New York : Dover Publications, Inc., 2015. - 309 p.
18. Сутула О.В. Переваги використання реактивної парадигми програмування для створення Android додатків Збірник матеріалів VI Міжнародної науково-практичної конференції молодих вчених / Сутула О.В. // "ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ: ЕКОНОМІКА, ТЕХНІКА, ОСВІТА '2015", 19-20 листопада 2015 року, Київ, НУБіП України. – К.: НУБіП України, 2015. – 323 с.