Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

магістра
(назва освітньо-кваліфікаційного рівня)

студента *Моссура Данила Євгенійовича*
(ПІБ)

академічної групи *122М-19-1*
(шифр)

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

на тему: *Методи, алгоритми та інформаційна технологія оптимізації синхронізації, керування та відтворення станів списків з посторінковим відтворенням*

_____*Д.Є. Моссур*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
розділів кваліфікаційної роботи				
спеціальний	Проф. Алексєєв М.О.			
економічний	Доц. Касьяненко М.О.			

Рецензент				

Нормоконтролер	Доц. Сироткіна О.І.			

Дніпро
2020

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
Завідувач кафедри
Програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 20 20 Року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *122 Комп'ютерні науки*
(код і назва спеціальності)

студенту *122М-19-1* *Моссуру Данилу Євгенійовичу*
(група) (прізвище та ініціали)

Тема кваліфікаційної роботи *Методи, алгоритми та інформаційна технологія оптимізації синхронізації, керування та відтворення станів списків з посторінковим відтворенням*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора НТУ «Дніпровська політехніка» від 22.10.2020 № 888-с

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – процес побудови та відтворення списків з посторінковими запитами протягом створення додатків.

Предмет досліджень – методи синхронізації, керування та відтворення серій посторінкових запитів при паралельному виконанні.

Мета роботи – підвищення ефективності та швидкості виконання типових завдань при побудові мобільних додатків.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні та оптимізації методів синхронізації, керування та відтворення серій посторінкових запитів..

Практична цінність результатів полягає в тому, що описаний у роботі метод дозволяє:

- Виявити неповність висунутих до додатку вимог ще на етапі огляду виконавцями завдання;

- Створити стандарт для проектної команди, що дозволяє позбутися часових затрат на комунікації стосовно деталей відтворення типових завдань.
- Значно зменшити часові затрати на програмування екранів, які містять списки з посторінковим відтворенням, за допомогою відтвореної у роботі бібліотеки.

4 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець)**
Аналіз теми та постановка задачі	01.09.2020-19.09.2020
Побудова крос-платформної програмної бібліотеки для вирішення задачі керування та відображення станів списку з посторінковими запитами даних	19.09.2020-1.11.2020
Експериментальні дослідження та детальний аналіз розробленого вирішення у порівнянні з існуючими альтернативними інструментами	1.11.2020-1.12.2020

Завдання видав _____ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Моссур Д.Є.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *1.09.2020 р.*

Термін подання кваліфікаційної роботи до ЕК *1.12.2020*

РЕФЕРАТ

Пояснювальна записка: 55 стор., 13 рис., 2 таблиці, 3 додатка, 25 джерел.

Об'єкт дослідження: процес побудови та відтворення списків з посторінковими запитами протягом створення додатків.

Предмет дослідження: методи синхронізації, керування та відтворення серій посторінкових запитів при паралельному виконанні;

Мета магістерської роботи: підвищення ефективності та швидкості виконання типових завдань при побудові мобільних додатків.

Методи дослідження. при вирішенні поставлених завдань виконано аналіз і наукове узагальнення програмних бібліотек, літературних джерел та персонального досвіду розробників мобільних додатків.

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні та оптимізації методів синхронізації, керування та відтворення серій посторінкових запитів.

Практична цінність результатів полягає в тому, що описаний у роботі метод дозволяє:

- Виявити неповність висунутих до додатку вимог ще на етапі огляду виконавцями завдання;
- Створити стандарт для проектної команди, що дозволяє позбутися часових затрат на комунікації стосовно деталей відтворення типових завдань;
- Значно зменшити часові затрати на програмування екранів, які містять списки з посторінковим відтворенням, за допомогою відтвореної у роботі бібліотеки;

У розділі «Економіка» проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПЗ і тривалості його розробки, а також проведені маркетингові дослідження ринку збуту створеного програмного продукту.

Список ключових слів: список з посторінковим відтворенням, типи посторінкових запитів, машина станів, мобільні додатки.

ABSTRACT

Explanatory note: 55 pages, 13 images, 2 tables, 3 appendices, 25 sources.

Objects of research: process of building pageable lists during application development.

Subject of research: methods of synchronization, managing and displaying asynchronous series of requests.

Purpose of the master's work: improving efficient and speed of implementation for typical tasks during mobile application development.

Methods of research: in the solution of the task set, analysis and scientific generalization of the programming libraries, literature and personal mobile developers' experience.

Scientific novelty: of the qualification work results is improvement of methods for synchronizing, managing and displaying page-by-page list requests series.

Practical significance: of the results help:

- To detect incomplete criteria for an application on the grooming step;
- To develop a standard for a project team, that allows to get rid of time expenses for communications about typical tasks details.
- To reduce time expenses for programming of screens that contain lists with page-by-page displaying via the library developed in the qualification work;

In the section "Economics" the calculations of the complexity of software development, the cost of creating software and the duration of its development, as well as marketing studies of the market for the created software product were conducted.

List of keywords: pageable list, page-by-page request types, state machine, mobile applications.

ЗМІСТ

ВСТУП

Актуальність роботи. Компанія IBM 23 вересня 1992 року представила світу перший смартфон. IBM Simon не набув широкого розповсюдження, але минуло 28 років і тепер смартфони продаються краще за настільні персональні комп'ютери. Смартфони зараз вже не просто інструменти для зв'язку, розрахунків, пошуку інформації тощо. Смартфони люди використовують і для ігор, відтворення мультимедійних файлів, розповсюдження новин та іншої інформації, фіксацій моментів свого життя на фото відео.

Таке розповсюдження смартфонів і широкий спектр їх використання притягує увагу бізнесу як ще один канал для збільшення аудиторії та покращення своїх сервісів. Бізнес створює попит для створення нових мобільних додатків. Вже існує багато компанії, які займаються створенням таких додатків.

Майже не можливо зараз знайти додаток, який не містив би хоча б один список. Бізнес, як і ринок, дуже змінливий, тому майже всі данні зберігаються не на пристроях користувачів, а на серверах. Списки – не виняток. Такі умови впливають на стани відображення, оскільки даних під час запуску додатку на пристроях немає, то поки він буде загружатися користувачу потрібно відобразити відповідний стан. Якщо це буде просто пустий екран користувач подумає що додаток не відповідає і може без зволікань установити додаток конкурентів і бізнес втратить клієнта. Таких станів декілька і вони розповсюджуються майже на кожен список.

Таким чином, ми маємо частину бізнес логіки, яка повторюється і начебто можна написати абстрактну реалізацію таких станів. Але існуючі бібліотеки створюють цілий ряд обмежень для їх використання на ряду з неможливістю змінити їх код тільки сповільнює процес розробки додатка.

Об'єкт досліджень: Процес побудови та відтворення списків з посторінковими запитами протягом створення додатків.

Предмет досліджень: методи синхронізації, керування та відтворення серій посторінкових запитів при паралельному виконанні.

Мета роботи: підвищення ефективності та швидкості виконання типових завдань при побудові мобільних додатків.

Наукова новизна результатів кваліфікаційної роботи полягає в удосконаленні та оптимізації методів синхронізації, керування та відтворення серій посторінкових запитів.

Практичне значення результатів полягає в тому, що описаний у роботі метод дозволяє:

- Виявити неповність висунутих до додатку вимог ще на етапі огляду виконавцями завдання;
- Створити стандарт для проектної команди, що дозволяє позбутися часових затрат на комунікації стосовно деталей відтворення типових завдань.
- Значно зменшити часові затрати на програмування екранів, які містять списки з посторінковим відтворенням, за допомогою відтвореної у роботі бібліотеки

Особистий внесок автора полягає в розробці теоретичної частини магістерської роботи, в дослідженні і систематизації знання про існуючі методики, розробці методів досліджень і технологій реалізації, в оцінці отриманих результатів.

Структура та обсяг кваліфікаційної роботи. Робота складається з вступу, трьох розділів і висновків. Містить 55 сторінок друкованого тексту, в тому числі 37 сторінок тексту основної частини з 13 рисунками, списку використаних джерел з 25 найменуваннями на **s** сторінках, 3 додатків на 8 сторінках.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ

1.1. Вимоги до типова задачі з відображення посторінкових списків

Більшість вимог, сформульованих для відтворення екранів з посторінковими списками враховують існування пустого стану, коли даних ще не існує або їх не вдалось отримати. Умова відсутності даних зазвичай виконується при старті публічного використання інформаційної системи. Саме тому для мобільного додатку стартовим станом має бути пустий стан (рис 1.1), який сповіщає користувача, про причину відсутності даних і дає можливість повторити запит на першу сторінку.

Рис 1.1. Побудова бізнес моделі станів екрану з посторінковим списком, крок 1

Коли пустий стан було відображено користувач може ініціювати повторний запит або він може виконатися автоматично. Запит на оновлення даних приведе користувача на екран пустої загрузки (рис 1.2), який відображує тільки прогрес-бар, оскільки інших даних ще не було отримано.

Вище згаданий запит сторінки може завершитися одним з чотирьох варіантів:

1. Сервер повернув усі данні для першої сторінки;
2. Сервер повернув пустий список (даних немає);
3. Сервер повернув не повну сторінку (це остання сторінка);
4. Запит завершився помилкою;

Кожна подія призведе до транзакції (рис 1.3). Перша змінить стан пустої загрузки відображенням списку (стан з даними). Друга поверне до попереднього, пустого стану. Третя приведе до стану з усіма даними. Четверта призведе до пустого стану з помилкою, оскільки жодних даних на пристрої немає, а запит завершився помилкою.

Рис 1.2. Побудова бізнес моделі станів екрану з посторінковим списком, крок 2

Пустий стан з помилкою повідомляє користувачу деталі помилки, можливо дає доступ до зворотного зв’язку, що вже не відноситься до поточного функціоналу. Тому за даного стану можна тільки створити запит на повторне виконання запиту, якщо проблема була тимчасовою або перешкода була на стороні клієнту.

Рис 1.3. Побудова бізнес моделі станів екрану з посторінковим списком, крок 3

Рис 1.4. Побудова бізнес моделі станів екрану з посторінковим списком, крок 4

Стан «Повні дані» описує стан, коли всі дані вже отримані та відображені. В такому випадку можна тільки створити запит на оновлення даних, що приведе користувача до пустого стану, якщо дані були видалені, стану з даними, якщо було повернуто першу сторінку, або знов до стану з повними даними, якщо запит на першу сторінку поверне усі існуючі дані. (рис 1.5)

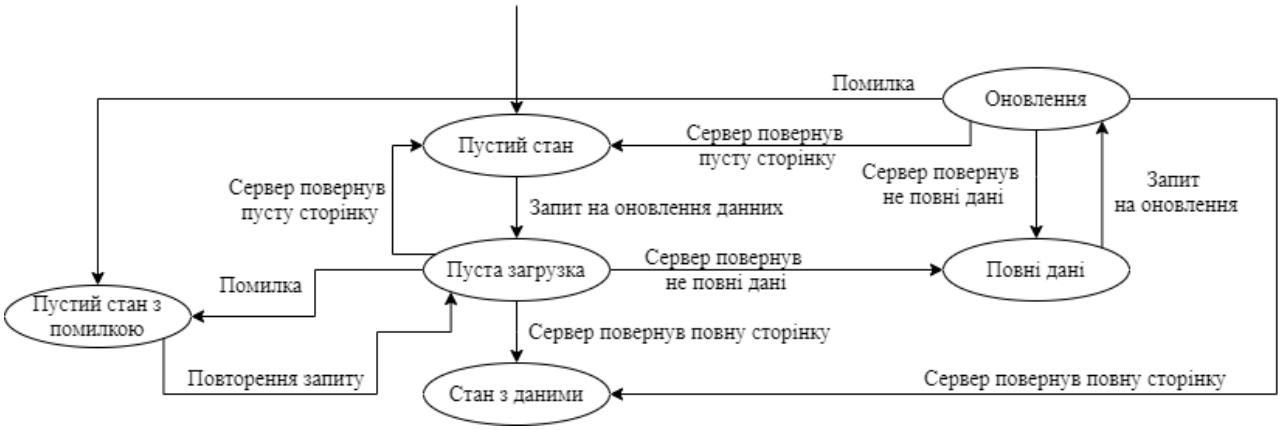


Рис 1.5. Побудова бізнес моделі станів екрану з посторінковим списком, крок 5

Стан з не повними даними дозволяє користувачу створити запит на наступну сторінку або запит на оновлення даних (рис 1.6).

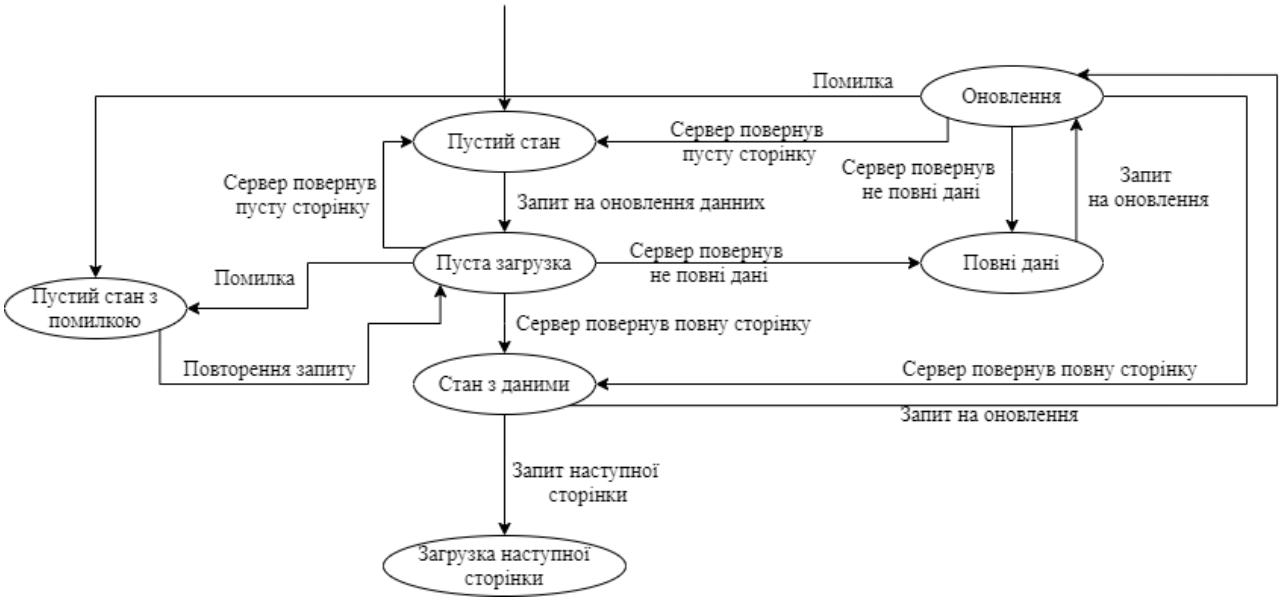


Рис 1.6. Побудова бізнес моделі станів екрану з посторінковим списком, крок 6

Запит на завантаження наступної сторінки може завершитися одним з чотирьох, наведених результатів, приведених вище для запита першої сторінки. Всі чотири результати призведуть до відповідних станів (рис 1.7).

Рис 1.7. Побудова бізнес моделі станів екрану з посторінковим списком, крок 7

1.2. Скінченний автомат

Створена в результаті аналізу області використання модель нагадує автомат, де запити від користувача або результати запитів до сервера складають мову автомату. Кількість згаданих станів для екрану відображення списку з посторінковими запитами скінченна, як і кількість подій які можуть змінювати ці стани також скінченна. Спираючись на це, можливо запропонувати, в якості рішення задачі, використання скінченний автомат.

Скінченний автомат, або як ще його називають у англомовній літературі, машина станів – це математична розрахункова модель, абстрактний автомат, який може знаходитись в одному зі скінченної кількості притаманних йому станів. Перехід з одного стану в інший називається транзакцією. Новий стан скінченного автомату буде обраний спираючись на вхідні дані, які викликали транзакцію та попередній його стан. [1] Скінченний автомат можливо записати у вигляді формули (1.1). Пам’ять даного автомата обмежена кількістю його станів.

$$FA=(Q, \Sigma,\delta,q_0,F) \tag{1.1}$$

де

Q – множина станів автомату;

Σ – алфавіт;

δ – функція переходу між станами;

q_0 – початковий стан автомату;

F – множина кінцевих станів автомату;

Алфавітом автомату називають скінченну множину значень, які називаються літерами. Автомат розпізнає послідовність літер, або слово, яку зчитує послідовно, літера за літерою. Якщо остання літера залишає автомат у

кінцевому стані, слово відноситься до мови автомату. Визначення автомату записують у формі формули (1.2).

$$L(FA) = \{ώ \in Σ^* | <умова>\} \quad (1.2)$$

де

L() – мова;

FA – назва автомату;

ώ – слово;

Σ – алфавіт автомату;

<умова> - умова, за якою слово визначається як слово мови автомата FA;

Для побудови та представлення скінченних автоматів використовують діаграми станів та таблиці формату Стан/Вхідний параметр. Таблиці описують функцію переходу між станами δ залежно від дійсного стану автомату та доданого вхідного параметру. Діаграма станів автомата – це направлений граф, у якому вершини відтворюють стани, а ребра – транзакції. Початковий стан автомату позначається ребром без початку. Кінцеві стани позначаються додатковим колом навколо вершини.

Скінченні автомати поділяють на детерміновані та не детерміновані. В детермінованих скінченних автоматах кожен вхідний параметр відповідає одній транзакцій. Для не детермінованих автоматів це правило не працює, і один вхідний параметр може призвести до однієї, декількох транзакцій або до жодної.[2]

1.3. Побудова автомату

Спираючись на аналіз вимог, приведений у розділі 1.1 та теоретичну частину розділу 1.2, побудуємо скінченний автомат для відображення станів списку з посторінковим завантаженням PFA (Pageable Final Automaton).

Спираючись на аналіз області застосування, приведений у розділі 1.2 можна побудувати множину станів автомату та записати її у формульному вигляді (1.3).

$$Q = \{Empty, EmptyProgress, EmptyError, Data, Refresh, NewPageProgress, FullData\} \quad (1.3)$$

де

Empty – пустий стан, який сповіщає користувача, що на пристрої немає даних для відображення та дозволяє повторити запит на першу сторінку;

EmptyProgress – повноекранний прогрес-бар, який вказує на перше завантаження даних;

EmptyError – цей стан описує помилку яка відбулась під час запиту та дозволяє повторити його;

Data – в цьому стані користувач бачить одну чи декілька повністю завантажених сторінок;

Refresh – окрім відображених вже наявних даних, на екрані присутній прогрес-бар, який вказує відображає стан оновлення даних;

NewPageProgress – у кінець сторінки додається прогрес бар завантаження нової сторінки;

FullData –всі доступні дані вже завантажені з сервера;

В якості вхідних параметрів скінченний автомат посторінкового відображення може отримувати події від користувача, через користувацький інтерфейс та події які сповіщають про результат запиту. Алфавіт автомату з умовними позначеннями подій відображений у формулі(1.4).

$$\Sigma = \{Restart, RefreshEvent, LoadMore, NewPage, PageError, Edit\} \quad (1.4)$$

де

Restart – запит на повне оновлення даних, який може бути викликаний при повторному відкритті екрану;

RefreshEvent – оновлення даних, яке може викликати користувач;

LoadMore – запит на наступну сторінку, може бути викликаним користувачем;

NewPage – результат завантаження наступної сторінки;

PageError – помилка при завантаженні сторінки;

Edit – зміна вже відображених даних, може бути викликана як користувачем, так і додатком у разі сигналу від серверу. Прикладом такої події може бути функція оновлення даних у реальному часі, такі як чати, сторінки з новинами, тощо;

Функція переходів між станами описана табл. 1.1. Функція переходів для автомату є повною, тож усі переходи є визначеними. Лівий стовбець таблиці описує дійсний стан автомату на момент додання нової події. Тип події вказано у першому рядку таблиці. Пусті клітинки вказують на те, що подія не буде оброблена та стан автомату залишиться незмінним.

Як вже було зазначено у розділі 1.3, початковим станом необхідно рахувати стан Empty, оскільки жодних даних ще не було ще отримано і в пам’яті автомату ще немає жодного стану з даними. Оскільки за кращими практиками проектування користувацького досвіду необхідно завантажувати дані одразу після переходу на екран. Перше завантаження буде викликано додатком спираючись на його стан.

Оскільки користувач може перервати роботу не тільки автомата, а й всього додатку у будь-який момент, то будь-який стан з множини станів автомату можна рахувати кінцевим. Таким чином мова скінченного автомата приймає всі можливі слова, складені з алфавіту Σ (1.5).

$$L(PFA) = \{ώ \in \Sigma^*\} \quad (1.5)$$

Таблиця 1.1

Таблиця функцій переходів скінченного автомата для екранів з посторінковими списками

	Restart	**Refresh Event**	**Load More**	**New Page**	**Page Error**	**Edit**
Empty	Empty Progress	Empty Progress				Data / Empty
Empty Progress				Data/ Empty	Empty Progress	NewPageProgress / EmptyProgress
Empty Error	Empty Progress	Empty Progress				
Data	Empty Progress		New Page Progress			Data / Empty
Refresh	Empty Progress	Refresh		Data/ Empty	Empty Progress	Refresh / EmptyProgress
New Page Progress	Empty Progress			Data/ Empty	Empty Progress	NewPageProgress / EmptyProgress
FullData	Empty Progress	Refresh				FullData / Empty

Спираючись на приведену вище аргументацію та використовуючи в якості бази описану за допомогою таблиці 1.1 функцію зміни станів, можливо побудувати граф станів скінченного не детермінованого автомату PFA, який зображено на рис. 1.8.

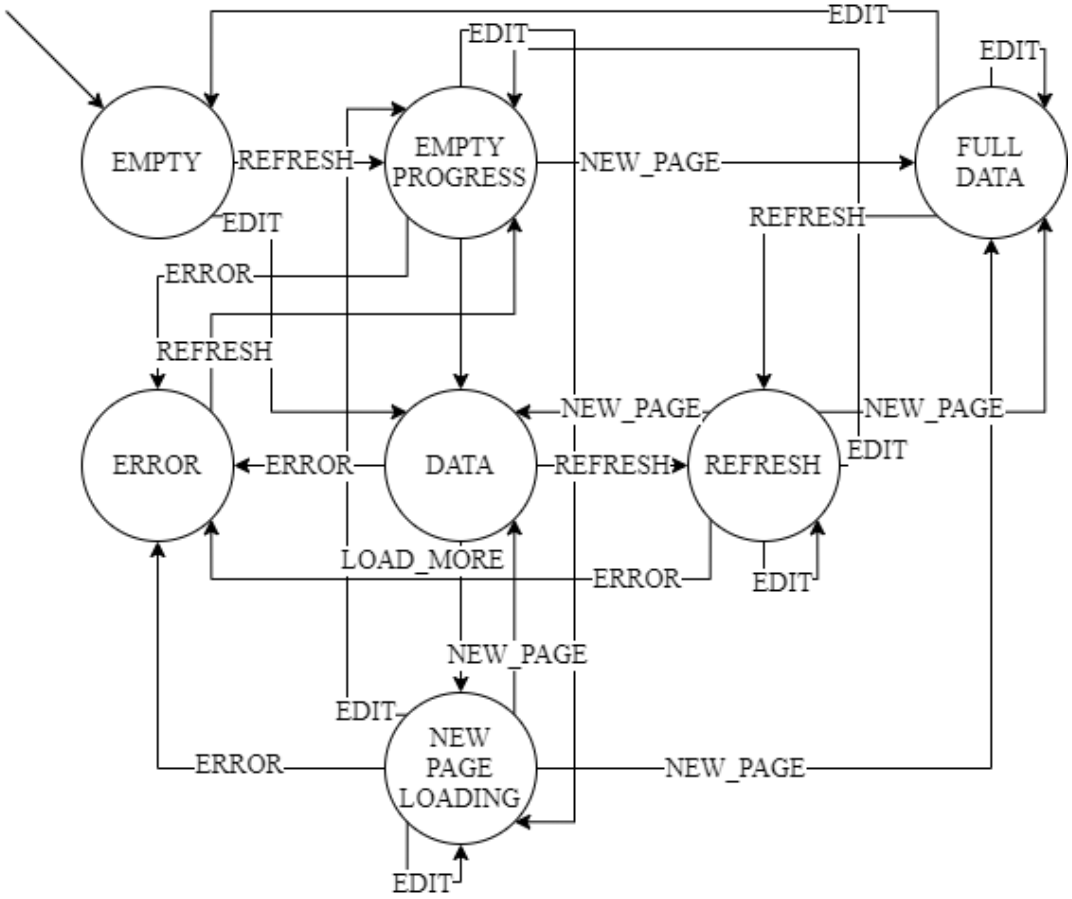


Рис 1.8. Граф станів скінченного автомату PFA

1.4. Алгоритм Маєрса

При доданні нового пакету даних до вже відображених необхідно провести порівняння старого та нового списку щоб визначити, які елементи було змінено або додано і відобразити вказані зміни на екрані. Також деякі види посторінкових списків потребують можливості змін у реальному часі. Наприклад, список діалогів у чат-клієнті, який відображає останнє отримане повідомлення для кожного відображеного чату, або список статей у новинному порталі, який змінює стан лічильників реакцій на статті, щойно один з користувачів натисне певний компонент. На графах станів скінченного автомату у розділі 1.3 можна знайти сингал Edit для таких випадків, який буде подаватися програмно спираючись на взаємодію користувача з інтерфейсом та на отримані події з серверу через сокет з’єднання.

Для вирішення зазначеної задачі необхідно визначити алгоритм дій з елементами двох списків, старого та нового, який дозволить максимально оптимальним шляхом визначити елементи, які потребують змін у відображенні, видалення чи додання в якості графічного компоненту. Найефективнішим алгоритмом такої спрямованості вважається алгоритм Маєрса для порівняння двох послідовностей[8]. Складність даного алгоритму у найпростішому випадку визначається як:

$$O(ND) \tag{1.6}$$

де

N – сума довжин порівнюваних послідовностей;

D – розмір мінімального сценарію редагування для порівнюваних послідовностей;

Відповідно до даних приведених у оригінальній статті Євгена Маєрса [8], очікуваний час виконання алгоритму для базової стохастичної моделі розраховується наступною формулою:

$$O(N+D^2) \tag{1.7}$$

Графічний метод втілення алгоритму передбачає побудову таблиці або координат в якості горизонтальних значень для якої взято елементи початкової послідовності, а в якості вертикальних значень – елементи кінцевої послідовності. На основі побудованої таблиці або системи координат можливо побудувати граф який описує алгоритм змін до початкової послідовності елементів щоб отримати кінцеву. Кожна точка в даній системі координат представляє можливу вершину майбутнього графу. Спираючись на дані позначення в таблиці або в системі координат буде побудовано граф, за яким можливо відстежити зміни у початковій послідовності.

У якості прикладу нижче буде приведено застосування алгоритму до таких послідовностей ABDDCD та DBDACB з побудовою таблиці.

Клітинки з однаковими елементами закреслюються з лівого верхнього кута у правий нижній. Далі будується граф, метою якого є досягти з лівого верхнього куту правий нижній з найменшою вагою. Ребра графа можуть бути направлені вправо, вниз або по діагоналі, яка була створена на попередньому етапі. Вага горизонтальних та вертикальних ребер по 1, а діагональних – 0.

Приведена нижче таблиця 1.2 зображує перший крок алгоритму, а саме визначення однакових елементів у вхідній та вихідній послідовностях і створення діагональних ребер за які будуть використані на наступному кроці побудови графу.

Таблиця 1.2

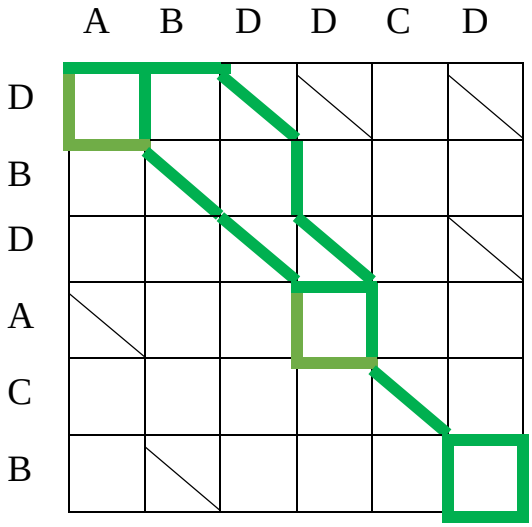
Корок перший алгоритму Маєрса. Таблиця відповідностей між елементами порівнюваних послідовностей.

	A	B	D	D	C	D
D			╲	╲		╲
B		╲				
D			╲	╲		╲
A	╲					
C					╲	
B		╲				

На таблиці 1.3 побудовано найкоротший граф для даних двох послідовностей. Основне правило заповнення, коли ребро торкається кута клітинки з діагоналлю, наступним ребром буде діагональ, вага якої 0. Вага горизонтального та вертикального ребра дорівнює 1. Тому розрахована вага побудованих графів дорівнює 6.

Таблиця 1.3

Крок другий алгоритму Маєрса. Таблиця з побудованим графом



Побудований за допомогою алгоритму Маєрса граф дозволяє визначити найбільш оптимальну послідовність змін до вхідного списку, які необхідно виконати щоб отримати вихідний список. При виконанні такого аналізу кожне горизонтальне ребро означає що відповідний елемент з початкової послідовності буде видалено, вертикальне ребро відповідає доданню відповідного елементу з нової послідовності, а діагональне ребро вказує на повну відповідність елементів, тож той самий елемент залишається на своєму місці.

1.5. Висновки до першого розділу

У даному розділі було виконано детальний аналіз бізнес вимог до користувацького інтерфейсу та користувацького досвіду екранів, які містять списки даних з посторінковими запитами. Для організації та представлення бізнес логіки в математичному вигляді було побудовано скінченний не детермінований автомат. В якості інструменту для оптимізації відображення послідовності з пакетів даних було запропоновано алгоритм порівняння строк Маєрса.

РОЗДІЛ 2
ПОБУДОВА БІБЛІОТЕКИ

2.1. Мова реалізації

Спираючись на власний користувацький та інженерний досвід, можливо зазначити, що поставлена задача розповсюджена для усіх типів платформ незалежно від операційної системи, версії та типу браузера, мови програмування, парадигми побудови UI компонентів, крос-платформного фреймворку тощо. Беручи до уваги таку різноманітність технологій, найраціональнішим рішенням представляється вибір широко розповсюдженої мови, яка орієнтована на крос-платформне застосування та легко інтегруються зі звичними для різних платформ мовами та інструментами, для втілення проекту. Найбільшого розповсюдження серед мов, які відповідають зазначеним вище критеріям, набули: Java, JavaScript, Dart, Kotlin.

Java для виконання на пристрої вимагає наявності інстальованої Java Virtual Machine. Програми написані на Java компілюються в Java Bytecode який виконується віртуальною машиною. Такі програми довше запускаються тому в більшості випадків при розробці надають переваги звичним для платформи рішенням, з якими найчастіше Java не має добре налаштованої інтеграції.

JavaScript використовують для написання додатків на базі фреймворку React Native. При використанні React Native JavaScript буде інтерпретованим вже на кінцевому пристрої, що сповільнює виконання додатку, потребує підключення досить великого фреймворку, що збільшує розмір додатку. Окрім фреймворку в JavaScript існує така проблема як доволі специфічна реалізація ООП що в рази зменшить читабельність коду, а динамічна типізація може призвести до фатальних помилок підчас виконання додатку.

Dart досить молода мова програмування, розроблена компанією Google для використання в першу чергу в парі з крос-платформним фреймворком Flutter,

який також належить до розробок Google. Крос-платформні фреймворки не користуються великим авторитетом серед інженерів, оскільки специфічні для платформ задачі доводиться вирішувати через додання нативного коду, що в свою чергу ускладнює розробку і створює кодову базу, яку важко підтримувати, що в свою чергу, сповільнює подальшу розробку ПЗ. Компанія Google продовжує працювати над Dart і робить його більше схожим на Kotlin щоб спробувати переманити Android інженерів до крос-платформної розробки.

Kotlin, на відміну від згаданих вище мов досить добре інтегрується для використання на різних платформах. Компанія JetBrains яка розробила Kotlin у серпні 2020 року представила Kotlin Mutliplatform Moblile (KMM). KMM – це крос-платформне SDK яке дозволяє компілювати Kotlin в JVM байткод для Android та Java сумісних додатків, або в LLVM для додатків для iOS [3]. Окрім KMM також існує технологія Kotlin/Native яка дозволяє компілювати Kotlin код для будь-якої платформи та викликати Kotlin код з таких мов як Java, JavaScript, C, Objective-C, Swift [14]. Така близька інтеграція за нативними інструментами збільшує швидкість запуску на відтворення додатків з використанням Kotlin на відміну від інших крос-платформних мов. Kotlin набув широкого розповсюдження завдяки визнанню Google цієї мови основною для розробки додатків для Android. Таким чином майже кожен Android інженер може підтримувати код на Kotlin. Розробка JetBraings має статичну типізацію, зручну і зрозумілу імплементацію ООП та багато інструментів, які оптимізують написаний код і спрощує його читабельність.

2.2. Відтворення машини станів

Використовуючи для подальшої роботи розроблені та побудовані у підрозділі 1.3 модель станів та мову скінченного автомату, можна побудувати ERR-діаграму класів для основної частини бібліотеки, метою якої є встановити базову логіку зміни станів та синхронізації відтворюваних списків. Основним

класом буде Paginator основною задачею якого є агрегація всіх сутностей які мають відношення до скінченного автомату. Таким чином вкладені класи Action та State відповідають мові автомату та множині його станів відповідно, а метод reducer зберігає функцію переходів станів скінченного автомату.

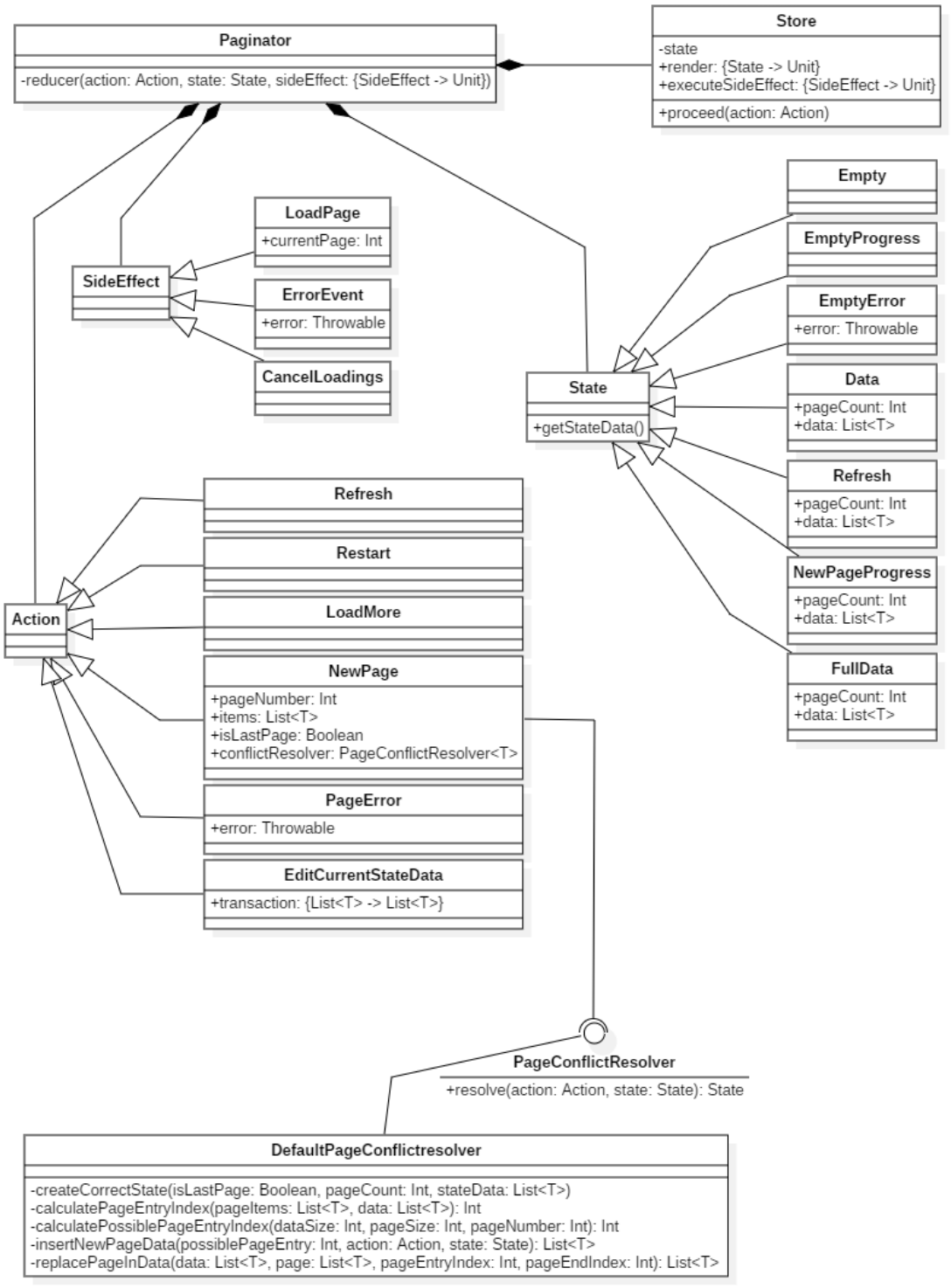


Рис. 2.1. EER-діаграма класів моделюючих скінченний автомат PFA

Виходячи із загального положення і користуючись в якості аргументу та логічного обґрунтування сукупністю раніше згаданих емпіричних знань саме функція переходів станів визначає коли необхідно завантажити наступну сторінку, обробити помилку чи відмінити дійсне завантаження даних, до класу Paginator необхідно додати утилітний клас SideEffect за допомогою якого можна сповістити оточення про необхідність обробки вище зазначених випадків.

Пам’ять скінченного автомату представлена об’єктом класу Store, який окрім стану зберігає функції для зворотного сповіщення оточення про зміну стану та необхідність обробки сайд ефекту.

Задля запобігання проблем касту об’єктів до певних типів, а також для більшої гнучкості до усіх методів та класів, які працюють зі списком даних було додано тип T, який обмежує можливі варіанти доданих даних. Такі типи також називають дженеріками або шаблонами. [5]

Відповідно до діаграми класів, список спадкоємців класів State, Action та SideEffect обмежений. Для таких випадків у Kotlin існує модифікатор класу sealed, який дозволяє обмежити список спадкоємців у коді для спрощення виразів перевірки на тип об’єкту та інкапсуляції можливості наслідування. Описаний вище модифікатор вказується перед ключовим словом class.[5] Спадкоємців класів об’явлених як sealed class дозволено створювати виключно у тому ж документі, де знаходиться їх предок. Задача визначення класу досить поширена в контексті даної архітектури Для визначення класу об’єкту використовується оператор when., на об'єктах sealed класів не потребується створення додаткової вітки else, яка описує випадок для не зазначений у попередніх вітках. Оскільки список спадкоємців лімітований, то кожна вітка оператора when буде відповідати певному класу. Лістинг для sealed класу State буде виглядати наступним чином:

```
sealed class State<T> {

    open fun getStateData(): List<T> = emptyList<T>()
```

```
        class Empty<T> : State<T>()
        class EmptyProgress<T> : State<T>()
        data class EmptyError<T>(val error: Throwable) : State<T>()
        data class Data<T>(val pageCount: Int, val data: List<T>) : State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class Refresh<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class NewPageProgress<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class FullData<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }
    }
```

Сповіщення оточення про подію за допомогою SideEffect буде реалізовано використовуючи лямбда-вираз, тобто анонімних функцію, яка зберігаються як змінна executeSideEffect в класі Store. Лямбда-вираз executeSideEffect буде викликано з об’єктом SideEffect як параметром.

2.3. Побудова системи для відтворення списків на UI.

У цьому підрозділі доречно звернути увагу на той факт, який вже був згаданий вище, що існує дві парадигми програмування системного UI: імперативний та декларативний. Імперативна парадигма є класичним рішенням для більшості платформ та легше піддається узагальненню логіки з відображення, оскільки вона передбачає представлення компонентів об’єктами. Декларативний UI для мобільних платформ знаходиться на експериментальній стадії і через велику кількість згенерованого коду представляється специфічним для кожної платформи. Цей факт робить неможливим створення одного модуля з логікою відображення списків. Перспективу для вирішення даної проблеми відкриває створення окремих модулів для кожної платформи та крос-платформених фреймворків. Покриття усіх можливих випадків не є метою даної

роботи, саме тому у цьому підрозділі буде втілене вирішення для ОС Android. Оскільки UI для Android додатків створюється за допомогою імперативного підходу, код з такого модулю можна змінити для інших платформ з імперативним UI з мінімальними витратами ресурсів.

Усю логіку з відображення для імперативного підходу можливо розділити на дві частини:

1. Налаштування відображення компонентів;
2. Налаштування відображення елементів списку;

2.3.1. Налаштування відображення UI компонентів

Дана частина відповідає за ввімкнення або вимкнення видимості окремих компонентів екрану, та оновлення відображених елементів списку даних. Спираючись на досвід інженерів мобільних додатків та особистий користувацький досвід, необхідно зазначити, що більшість графічних та UX дизайнерів, як українських так і зарубіжних, часто створюють екрани додатків не звертаючись до створених та/або визнаних компаніями власниками платформ гайдлайнів, через специфічні вимоги бізнесу. Спираючись на цей факт, необхідно створити умови для максимальної гнучкості задання порядку розташування графічних компонентів на екрані. Перспективи у вирішенні даної проблеми відкриває відокремлення логіки з відображення від UI компонентів у делегат, який буде отримувати компоненти як параметри.[6] На EER-діаграмі класів, зображеній на рис. 2.2, такий делегат відображений як PaginatorViewDelegate.

Делегат з відображення маніпулює рядом специфічних UI компонентів, які передаються в конструктор та зберігаються в ньому доки життєвий цикл базових компонентів системи не знищить їх. Такими компонентами є спадкоємці інтерфейсів:

1. AbsPaginatorEmptyView – цей компонент сповіщає користувача про відсутність даних через пусту базу даних або через помилку;
2. AbsPaginatorView – компонент, який створений для зменшення повторень в коді шляхом однієї стандартної імплементації даного інтерфейсу та/або вирішення проблем з життевим циклом базових компонентів Android;

Окрім зазначених вище бібліотечних компонентів PaginatorViewDelegate включає стандартні компоненти які входять до складу Android SDK:

1. RecyclerView – компонент який використовується для відображення списків даних та запобігання помилки OutOfMemoryException. Для використання цього компоненту необхідно створити свого спадкоємця класу RecyclerView.Adapter, який відповідає за побудову списку з компонентів які будуть відображені відповідно до правил прописаних у спадкоємці RecyclerView.ViewHolder для відповідного йому елементу списку даних [7];
2. SwipeRefreshLayout – компонент який зчитує так званий overscroll event, коли користувач долистав до початку списку і продовжує листати далі, цей компонент викликає оновлення даних і відтворює відповідний стан візуально;
3. View – стандартний компонент Android. В даному випадку представляє компонент який відповідає за відображення повного оновлення списку з даними;

Оскільки базовий клас RecyclerView.Adapter не має методу, який відповідає за оновлення даних та не може відтворювати різні компоненти в одному списку, до моделі було додано AbsPaginalAdapter який наслідує імплементацію адаптера з бібліотеки AdapterDeleagates. Таке рішення є тимчасове оскільки вже взимку 2020-2021 років компанією Google буде представлений оновлений стандартний компонент, який вже відповідає всім вимогам даної абстракції.

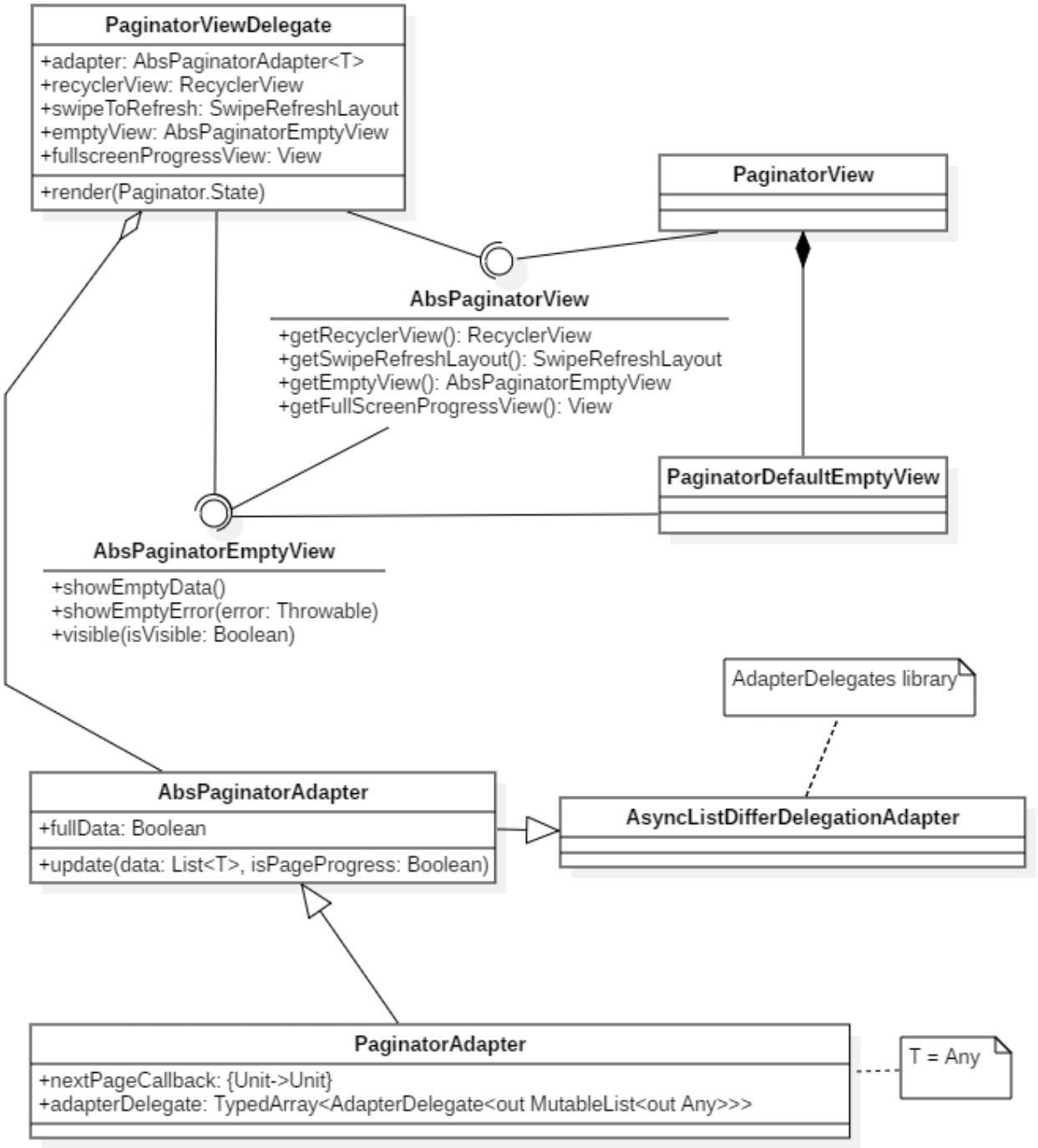


Рис. 2.2. EER-діаграма класів системи відображення станів скінченного автомату

2.3.2. Налаштування відображення елементів списку

Власні спостереження і спеціальні дослідження в плані створення та застосувань списків з посторінковим відтворенням показали, що екрани з посторінковим відтворенням можуть вимагати відтворення декількох типів

елементів одночасно, що у свою чергу є ще одною умовою. Окрім відображення елементів різних типів, ще одним обмеженням є розмір списку та умови його оновлення, оскільки оновлювати усі UI компоненти на екрані – досить затратна операція, необхідно відсіювати елементи які не змінились і не потребують змін у відображенні.

Виходячи із сукупності всіх раніше згаданих емпіричних знань в одній структурі можуть міститися об’єкти різних класів які відповідають різним графічним компонентам. Наприклад, список з коментарів може містити об’єкти спадкоємців одного класу Коментар і використовувати його визначення методів equals та hashCode для порівняння, але серед звичайних коментарів можуть міститися відповіді, для яких необхідно змінити UI компонент. Необхідно визначати відповідний компонент спираючись вже на тип і дані об’єкту, а не тільки на результат equals. Такі випадки покриваються за допомогою шаблону проектування обгортка, включеного до переліку шаблонів проектування банди чотирьох. Методи необхідні для даного рішення описані інтерфейсом AbsPaginatorItem.[6] Спадкоємці представленого інтерфейсу зможуть визначати логіку порівняння елементів списку для якого їх було застосовано.

Другу умову – оптимізацію оновлення списку виконують завдяки порівнянню безпосередньо даних, які необхідно відобразити між собою. Логіку такого порівняння елементів списку інкапсулює в собі стандартний інструмент Android SDK – DiffUtil. Зазначений інструмент використовує алгоритм Маерса, описаний в розділі 1.4 для пошуку елементів які змінилися та потребують оновлення відображених даних, далі по створеним алгоритмом змійкам DiffUtil визначає елементи які потребують видалення, додавання до відображених або зміни позиції. Методи необхідні для порівняння даних також додані до інтерфейсу AbsPaginatorItem.

Завдання класу AdapterDelegate – співвіднести елементи списку за їх типом та рядом вторичних значень з відповідними спадкоємцями класу

RecyclerView.ViewHolder [9]. Це факт приводить нас до висновку, що усі без винятку спадкоємці класу AdapterDelegate повинні виконувати наступні функції:

1. Визначати чи може створюваний делегатом ViewHolder відобразити переданий в нього елемент списку;
2. Створити ViewHolder та передати у нього UI компонент;
3. Передати у відповідний ViewHolder елемент без обгортки за наявності такої.

Спираючись на той факт, що AdapterDelegate та його спадкоємці не виконують інших, окрім зазначених вище, функцій у відтворенні елементів списку на UI, створювати відкритий для спадкування клас не є раціональним рішення. Поставлену задачу можна вирішити за допомогою фабричного методу, який буде повертати анонімний клас з вбудованими у нього підчас компіляції лямбда виразами. Клас PaginatorAdapterDelegatFactory втілює такий фабричний метод. Схожа ситуація для спадкоємців DiffUtil.ItemCallback. Для створення таких об’єктів фабричний метод втілений у PaginatorDiffItemCallbackFactory. Обидві фабрики мають додатковий метод для створення відповідних об’єктів для обробки стандартної імплементації обгортки – DefaultPaginatorItem, що значно зменшує кількість строк коду необхідних для створення середніх за складністю екранів. Задля наочності нижче приведено лістинг для класу PaginatorDiffItemCallbackFactory:

```
object PaginatorDiffItemCallbackFactory {

    val forPaginatorItem by lazy {
        create<AbsPaginatorItem<*>>(
            { oldItem, newItem -> oldItem.areItemsTheSame(newItem) },
            { oldItem, newItem -> oldItem.getChangePayload(newItem) }
        )
    }

    inline fun <reified T : Any> create(
        crossinline areItemsDataTheSame: (oldItem: T, newItem: T) -> Boolean,
        crossinline getChangePayload: (oldItem: T, newItem: T) -> Any = { _, _ -
> Any() }
    ) = object : DiffUtil.ItemCallback<Any>() {

        override fun areItemsTheSame(oldItem: Any, newItem: Any): Boolean =
            if (oldItem === newItem) {
                true
            } else if (oldItem is T && newItem is T) {
```

```
                areItemsDataTheSame(oldItem, newItem)
            } else {
                false
            }

        override fun getChangePayload(oldItem: Any, newItem: Any) =
            if (oldItem is T && newItem is T) {
                getChangePayload(oldItem, newItem)
            } else {
                Any()
            }

        override fun areContentsTheSame(oldItem: Any, newItem: Any) =
            oldItem == newItem
    }

}
```

В приведеному вище лістингу необхідно виділити що клас PaginatorDiffItemCallbackFactory втілює шаблон проектування singleton [6], що визначається у Kotlin за допомогою ключового слова object. Наявність одного об’єкта зменшує кількість необхідної для функціювання фабричного методу пам’яті та процесорного часу. Оскільки для залежності forPaginalItem буде створений один об’єкт, який можна використовувати в усіх випадках, коли необхідно додати порівняння класів обгорток, які наслідують бібліотечний інтерфейс AbsPaginatorItem. Разом з цим метод create є статичним методом і для його використання через наявність єдиного об’єкту немає необхідності створювати велику кількість власних класів та стільки ж об’єктів для них. В купі це сильно зменшує необхідний об’єм heap пам’яті необхідної для функціювання додатку.

Модифікатор inline перед методом create позначає анонімні лямбда вирази для компілятора як вирази, які необхідно вбудувати в місце їх виклика, що значно скорочує час на обробку даного коду та пам’ять необхідну для виконання такої підпрограми. Разом з тим лямбда вирази позначені як crossinline тобто вони не можуть бути перервані під час свого виконання та повинні обов’язково повернути результат.

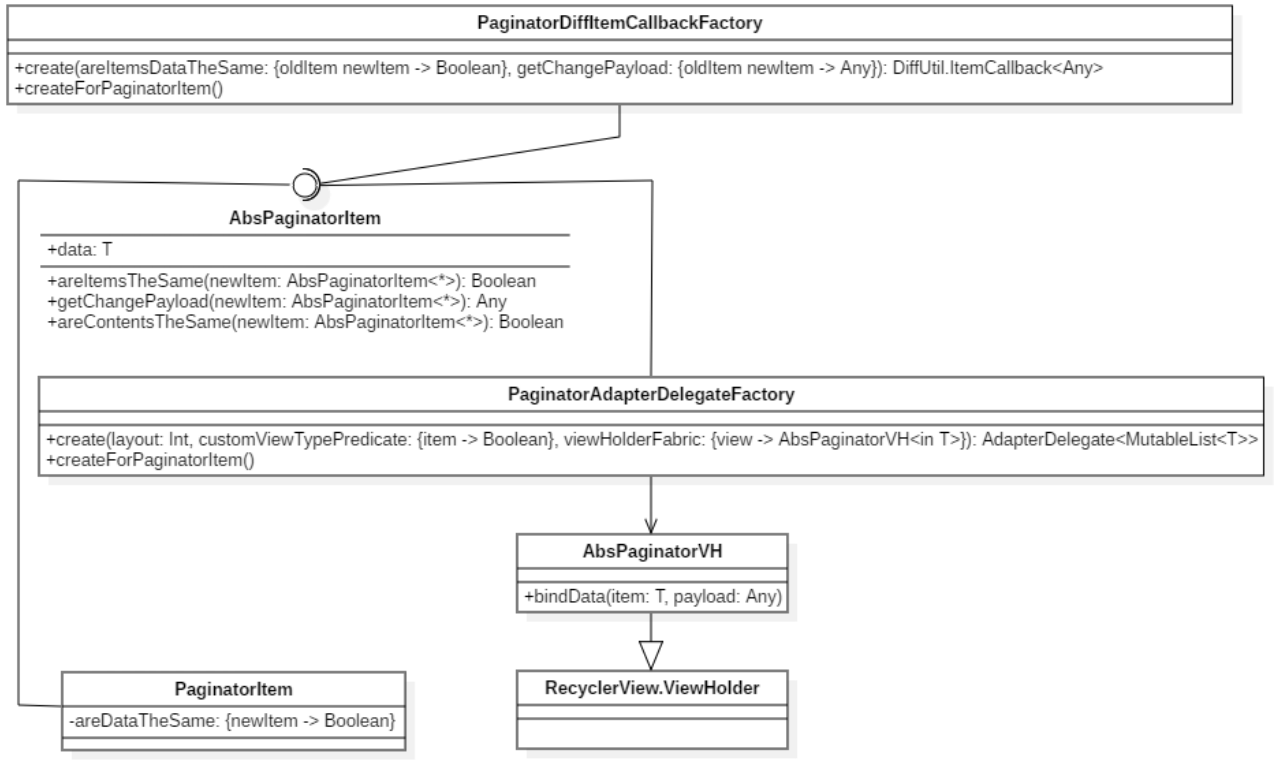


Рис. 2.3. EER-діаграма класів системи відображення елементів посторінкового списку на базі ОС Android

2.4. Висновки другого розділу

У розділі 2 було проаналізовано можливі крос-платформні інструменти, та використовуючи кращий з них, Kotlin Multiplatform Mobile, втілено у вигляді програмної бібліотеки розроблене, на базі отриманих в розділі 1 результатів, вирішення.

РОЗДІЛ 3
ЕКСПЕРИМЕНТАЛЬНІ ДОСЛІДЖЕННЯ

3.1. Пошук альтернативних вирішень

Спираючись на проведене експрес дослідження та аналіз готових рішень для оптимізації створення списків з посторінковими запитами для ОС iOS, можна зробити висновок, що всі представлені бібліотеки оптимізують лише частину з відображення списку та стану завантаження наступної сторінки і не враховують наявність інших станів екрану приведених у розділі 1.2. Фактично дані вирішення спрощують лише створення малої частини UI логіки разом з тим накладаючи ряд обмежень на гнучкість системи.

Схожа ситуація з бібліотеками для розробки під ОС Android. Пошук за ключовими словами та фразами на спеціалізованому ресурсі Android Arsenal [10] повертає або вирішення для вузького спектру станів, такі як для ОС iOS, або пустий результат, що вказує на відсутність подібних за своєю структурою вирішень для даної платформи. Єдиним відомим архітектурним вирішенням є Paging Library від компанії Google[11], але і ця бібліотека лише частково покриває необхідні кейси. Основною її перевагою є покриття усіх відомих типів посторінкових запитів інформації. Вказану особливість можна розглядати і як недолік, оскілки такий підхід потребує одночасного застосування на всіх трьох рівнях архітектури сучасних інформаційних систем: доменному, презентаційному та дата рівнях. А оскільки доменний рівень за своїм визначенням не повинен залежати від зовнішніх модулів, враховуючи бібліотеки, то такий підхід не буде канонічним і, як результат, впливатиме на побудову всієї системи.

Альтернативних бібліотек та сніппетів для крос-платформних технологій ще менше ніж для платформних через не велику популярність даних технологій серед профільних інженерів. Не зважаючи на невелику вибірку доступних

результатів, вдалося знайти бібліотеку moko paging [12] для Kotlin Multiplatform Mobile, створену невеликою аутсорсінговою компанією IceRock. На відміну від приведених вище бібліотек, moko paging – комплексне вирішення, яке покриває ряд необхідних для посторінкового списку станів, це робить її основною та єдиною альтернативою розробленому в розділі 2 вирішенню.

3.2. Дослідження альтернативного вирішення

Детальний аналіз альтернативного вирішення показує, що для використання модулю з посторінковими списками необхідно додати у проект ще одну залежність для модулю moko-mvvm. Такий зв'язок зобов'язує використовувати MVVM шаблон для побудови логіки додатку. Окрім цього, стани передаються виключно за допомогою реактивного підходу, а саме використовуючи в якості ресурсу даних LiveData, що може привести до конфліктів з компонентами інших поширених бібліотек для мобільної розробки. У випадку з розробкою додатку для ОС Android конфлікт буде створювати клас LiveData з поширеного в середі профільної розробки набору інструментів Android Jetpack, який імплементує той самий підхід, що й компонент moko-mvvm.

Після додання залежності від mvvm модулю, у проекті з'являється клас State і його спадкоємці. Для більш детального аналізу необхідно розглянути лістинг цих класів

```
sealed class State<out T, out E> {
    data class Data<out T, out E>(val data: T) : State<T, E>()
    data class Error<out T, out E>(val error: E) : State<T, E>()
    class Loading<out T, out E> : State<T, E>()
    class Empty<out T, out E> : State<T, E>()

    fun isLoading(): Boolean = this is Loading
    fun isSuccess(): Boolean = this is Data
    fun isEmpty(): Boolean = this is Empty
    fun isError(): Boolean = this is Error

    fun dataValue(): T? = (this as? Data)?.data

    fun errorValue(): E? = (this as? Error)?.error
}
```

З визначених у першому розділі семи станів, необхідних для коректного відтворення екрану з посторінковим списком у даному вирішенні присутні тільки чотири, що значно погіршує UX і може привести до значних затрат часу на відтворення додаткового стану при масштабуванні вимог до ПЗ. Зважаючи на розташування даної ієрархії та на тип T, який не обмежує його як спадкоємців списку та не є типом елементу списку, можна зробити висновок, що така структура була створена для відображення будь-якого екрану, що є більш абстрактним вирішенням, але у той самий час ускладнює обробку біль складних задач які потребують більшої кількості станів. Окрім недостатньої кількості станів у даній структурі вказано відповідний каст-метод для кожного класу, що знецінює можливості sealed класу і робить існування класів-спадкоємців надлишковим.

Під час розробки великих додатків велику роль в обранні інструментів та бібліотек для розробки відіграє розмір бібліотеки, оскільки для розповсюдження додатку використовуються магазини додатків, кожен з яких накладає обмеження на максимальний обсяг установочного файлу. Наприклад у Google Play Market максимальний розмір APK-файлу 50Mb, Apple App Store дозволяє завантажувати додатки розміром до 55Mb.

Для порівняння розмірів бібліотек після компіляції, за відсутності інструментів для компіляції iOS додатків, було використано плагін аналізу APK-файлів для Android Studio. На рисунках 3.1 та 3.2 відтворено інтерфейс плагіну з виділеними директоріями бібліотек PaginatorRenderKit та Moko-paging відповідно. Тут необхідно звернути увагу на кількість залежностей під одним доменним ім’ям. Під доменом dev.icerock.moko є ще один репозиторій, на який необхідно звернути увагу, а саму mvvm. Без зазначеної залежності використовувати Moko-paging неможливо, тому у порівняльній таблиці 3.1 для вирішення moko вказано генералізовані дані.

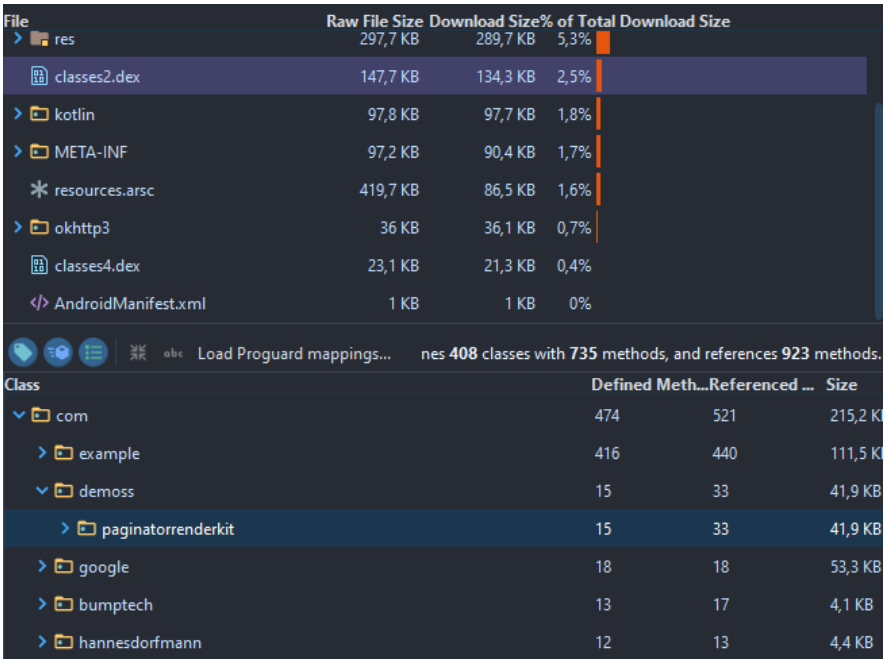


Рис 3.1. Аналіз APK файлу для Sample проекту з бібліотекою PaginalRenderKit

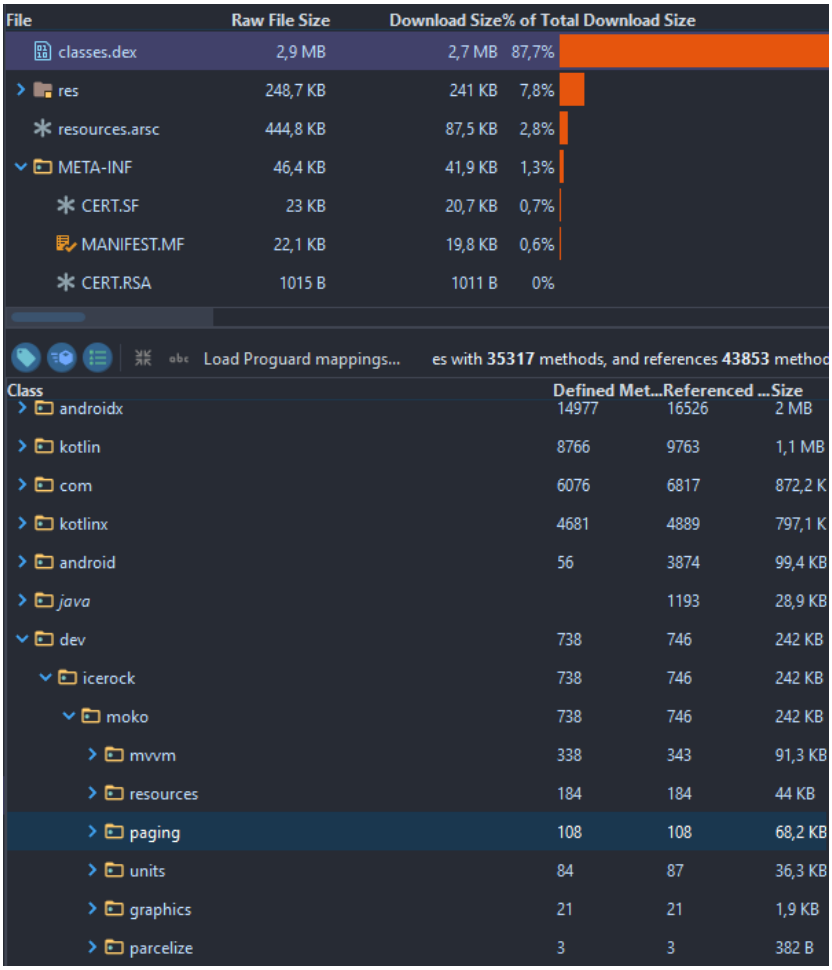


Рис 3.2. Аналіз APK файлу для Sample проекту з вирішенням Moko-paging + Moko-mvvm

Приведений вище аналіз виконаний на sample проектах до вказаних бібліотек. Для проекту з вирішенням від IceRock вимкнена мінімізація ресурсів для релізних варіантів побудови, при ввімкненні мінімізації стає можливим отримати помилку під час виконання програми, що погіршує UX додатку. А з вимкнутою мінімізацією увесь вихідний код проекту увійде у фінальний архів, що збільшує розмір додатку не тільки за рахунок розміру бібліотеки, а також і через невикористаний код, який розробник не видалить перед запуском побудови додатку.

Таблиця 3.1

Таблиця порівняння розробленого вирішення PaginalRenderKit з альтернативним Moko

	PaginalRenderKit	**Moko**
Розмір модуля з вихідним кодом	25.9Kb	14.8Kb + 71.6Kb
Розмір бібліотеки після компіляції	41.9 Kb	68.2 Kb + 91.3 Kb
Переваги	Логіка зі зміни станів та їх відображення розподілена між класами, які втілюють принцип єдиної відповідальності та інверсії залежностей[6], що дозволяє вносити зміни до сценаріїв використання без зайвих зусиль.	Представлені модулі містять базові класи, наслідування яких значно зменшують кількість коду необхідного для створення екрану з відповідним функціоналом, якщо вимоги до ПЗ не передбачають змін до вже розробленої та представленої у базових імплементаціях логіки.

Продовження таблиці 3.1

Обмеження при використанні	Бібліотека написана з використанням інтерфейсів для максимальної гнучкості. Єдиною незмінною складовою є функція скінченного автомату, оскільки ця функція втілює незмінну бізнес логіку зміни станів екрану	1. Для використання даного вирішення необхідно додати залежність від moko-mvvm, що збільшує кінцевий розмір додатку; 2. Додана залежність спонукає використовувати MVVM шаблон як основний; 3. Наявні в mvvm модулі класи можуть конфліктувати з сторонніми вирішеннями для різних платформ; Описані в вирішенні стани не покривають усіх необхідних випадків;

Спираючись на детальне порівняння розробленої у розділі 2 бібліотеки та існуючого вирішення від IceRock за значущими у розробці ПЗ критеріями, які приведено у таблиці 3.1, можна зробити висновок, що розроблене вирішення, на відміну від вже існуючого, є більш мобільним та гнучким, що значно збільшує вибірку проектів у яких можна застосувати дану бібліотеку без змін в архітектурі ПЗ.

3.3. Висновки до третього розділу

В ході роботи над третім розділом було розглянуто альтернативні вирішення поставленої задачі як для платформних так и для крос-платформних підходів. Серед знайдених альтернатив було виділено вирішення, яке найбільш близьке до вирішення задачі та покриває найбільшу кількість кейсів. Детальний аналіз та порівняння існуючого інструмента з розробленим у розділі 2 вказує на ряд переваг розробленої бібліотеки над існуючими.

РОЗДІЛ 4
ЕКОНОМІЧНА ЧАСТИНА

4.1. Маркетингові дослідження

Орієнований на масового споживача бізнес витрачає великі суми на рекламу, продаж та транспортування своїх товарів чи послуг за допомогою інформаційних технологій. Прикладом таких витрат є банери на інтернет сайтах, профільні мобільні додатки чи настільних застосунки, а також реклама в інформаційних та або соціальних групах. Одним з найефективніших та дешевих способів продавати послуги чи товари є використання широко розповсюджених e-commerce платформ чи створення свого додатку, який дозволить у карантинні часи надати послугу споживачеві дистанційно, чи назначити та або перенести зустріч на певний час для запобігання створення черг та зменшення ризику розповсюдження вірусу. Такий попит обумовлює існування та динамічний ріст IT сектору.

В якості розробника ПЗ може виступати стартап, продуктова чи аутсорсингова компанія. Незалежно від внутрішньої організації компанії-розробника, бізнес потребує швидких змін. Швидкість внесення змін чи додання нового функціоналу до вже працюючого проекту коштує великих грошей оскільки для забезпечення достатньої якості за короткий термін доводиться наймати високо кваліфікованих інженерів, заробітна платня яких може досягати десяти тисяч доларів а інколи і більше.

Одним з найрозповсюдженіших типів функціоналу ПЗ це створення та відображення списків. Це може бути список товарів, опцій або послуг, більшість з них проходить через ряд станів не очевидних для бізнесу, але важливих для користувача. Такі стани максимально інформують користувача про те, що відбувається на його пристрої, це справляє на нього враження безпечності та якості ПЗ, яким він користується. Таке враження називають user experience (UX),

або користувацький досвід. Чим кращий UX у додатка, тим довше і з більшим задоволенням користувач буде звертатися за послугою чи товаром до такого ПЗ.

4.2. Економічний ефект

Оскільки в ході даної роботи було створено програмну бібліотеку, яка може бути використана в парі з різними технологіями та розповсюджується за ліцензією Apache License 2.0, жодної плати за її використання не стягується. Різноманітність технологій, умов проектів та бізнес моделей компаній, які можуть використовувати даний продукт обумовлюють неймовірну складність розрахунків витрат до та після впровадження ПЗ, якщо такі розрахунки взагалі можливі. Через це розглядається лише соціальний ефект від впровадження в процес розробки такої бібліотеки.

В рамках кваліфікаційної роботи було розглянуто ряд згаданих у підрозділі 4.1 станів списків з посторінковим відтворенням та умови переходів між ними. Детальний аналіз найкращих UX практик дозволив створити стабільний алгоритм переходів між станами списків з посторінковим відтворенням. Створена бібліотека, яка базується на згаданому алгоритмі, в рази зменшує часові витрати на створення екранів зі списками, які є розповсюдженою і часто повторюваною задачею для кожного ПЗ, та її використання не потребує специфічних навиків. Використання одного коду для вирішення схожих завдань підвищує якість виконання та значно зменшує вірогідність впливу людського фактору на кінцевий результат. Таке вирішення зменшує необхідність найму висококваліфікованих розробників на проекти, які не містять специфічного функціоналу, що в свою чергу зменшує вартість команди розробки. Разом зі зниженням часу необхідного на втілення бізнес завдань зменшується і вартість розробки в цілому. Разом ці фактори складають мрію будь-якого бізнесу, проект втілюється швидше, якісніше та дешевше.

ВИСНОВКИ

Основною метою кваліфікаційної роботи було створення максимально ефективного інструменту для вирішення типових задач з відтворення екранів які містять списки з посторінковими запитами. В процесі роботи було розроблено бібліотеку, яка містить скінченний автомат станів списку та оптимізує логіку з відображення станів та елементів списків.

Функціональна частина бібліотеки розроблена на мові Kotlin, а скріпт модуля на Groovy для системи автоматичного збирання проектів Gradle, що дозволяє використовувати дане вирішення для крос-платформних проектів з використанням технології Kotlin Mutltiplatform Mobile. Для короткої документації проекту було застосовано мову MarkDown.

Протягом роботи над кваліфікаційною роботою було виконано наступні завдання:

1. Узагальнений аналіз бізнес вимог до типових задач з відображення екранів зі списками;
2. Побудовано скінченний автомат для станів екрану зі списком з посторінковими запитами;
3. Досліджено та застосовано алгоритм Маєрса для оптимізації відображення елементів посторінкового списку;
4. Розроблено крос-платформну бібліотеку;
5. Досліджено альтернативні вирішення;
6. Проведено детальний аналіз та порівняння найкращого альтернативного вирішення з розробленою бібліотекою;

В економічній частині було проведено маркетингові дослідження та визначено економічний ефект від створеного вирішення. Вартість розробленої бібліотеки визначити не вдалося через open source характер розповсюдження.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Yan, Song Y. (1998). An Introduction to Formal Languages and Machine Computation. / Yan, Song Y. - Singapore: World Scientific Publishing Co. Pte. Ltd. - 1998 - P. 155–156.

2. Brutscheck, M., Berger, S., Franke, M., Schwarzbacher, A., Becker, S. (2008). Structural Division Procedure for Efficient IC Analysis. IET Irish Signals and Systems Conference, P.18–23.

3. Roman Elizarov, Kotlin Mutliplatform Mobile [Електронний ресурс] / Roman Elizarov – режим доступу: https://kotlinlang.org/lp/mobile/

4. Soshin Alexey. Hands on design patterns with Kotlin / Soshin Alexey - Birmingham Packt - 2018 – P. 357.

5. Dmitry Jemerov, Svetlana Isakova. Kotlin in Action Version 11 / Dmitry Jemerov, Svetlana Isakova - New York City Manning Publications 2016 – 360 P

6. Erich Gamma and others. Design Patterns Elements of Object-Oriented Software / Erich Gamma and others - Addison-Wesley 1994 – P. 416

7. Google LLC. Recycler View Documentation [Електронний ресурс] / Google LLC – режим доступу : https://developer.android.com/guide/topics/ui/layout/recyclerview

8. Eugene W. Mayers. An O(ND) Algorithm and It's Varitations [Електронний ресурс] / Eugene W. Mayers – P – 4.

9. Mark L. Murphy (2018) Android Architectue Components / Mark L. Murphy - USA Commons Ware – P. 234.

10. Hannes Dorfmann. Joe's great Adapter hell escape [Електронний ресурс] / Hannes Dorfmann – режим доступу: http://hannesdorfmann.com/android/adapter-delegates

11. Mike McQuaid (2015) Git in Practice New York City Manning Publications– P. 225

12. Aleksey Mikhailov, Vladislav Areshkin. Moko-paging repository [Електронний ресурс] / Aleksey Mikhailov, Vladislav Areshkin. – режим доступу: https://github.com/icerockdev/moko-paging

13. Robert C. Martin and others (2008). Clean Code A Handbook of Agile Software Craftmanship / Robert C. Martin and others - Prentice Hall Boston – 2008 – P. 464.

14. Roman Elizarov, Kotlin Native Documentation [Електронний ресурс] / Roman Elizarov – режим доступу: https://kotlinlang.org/docs/reference/native-overview.html

15. Rivy Chakraborty (2017) Reactive Programming in Kotlin / Rivy Chakraborty - Birmingham Packt – P. 322.

16. Ted Hagos. Learn Android Studio 3 Efficient Android App Development / Ted Hagos - Manila Apress 2018 – P. 260

10. Android Arsenal resource [Електронний ресурс] – режим доступу: https://android-arsenal.com/

18. Ian F. Darwin. Android Cookbook 2nd Edition / Ian F. Darwin - USA O'Reilly Media Inc 2017 – P. 768.

19. Chad Fowler (2009) Passionate Programer 2nd edition / Chad Fowler Pragmatic Bookshelf – 2009 – P. 232.

21. Jirí Adámek and Věra Trnková. (1990). Automata and Algebras in Categories. / Jirí Adámek and Věra Trnková. - Kluwer Academic Publishers:Dordrecht and Prague

20. Martin Fowler (2000) Refactoring: Improvement the Design of Existing code / Martin Fowler - Boston Person Education Inc, Addison Wesley Longman – P. 419.

22. Google LLC. Android Jetpack Paging Architecture [Електронний ресурс] / Google LLC – режим доступу RL: https://developer.android.com/topic/libraries/architecture/paging

23. Roland Kuhn, Brian Hanafee, Jamie Allen (2017) Reactive Design Patterns / Roland Kuhn, Brian Hanafee, Jamie Allen - New York City Manning Publications – P. 351

24. ДСТУ ISO 9000:2007 Системи управління якістю. Основні положення та словник термінів. – К.: Держспоживстандарт України, 2008.

25. М.О. Алексєєв, О.І. Сироткіна, І.М. Удовик, О.С. Шевцова, Методичні рекомендації до виконання магістерських робіт студентами спеціальностей 121 «Інженерія програмного забезпечення» та 122 «Комп'ютерні науки» / М.О. Алексєєв, О.І. Сироткіна, І.М. Удовик, О.С. Шевцова; М-во освіти і науки України, Нац. техн. ун-т "Дніпровська політехніка". – Дніпро : НТУ «ДП», 2018. – № 3. – 57 с

ДОДАТОК А

Лістинг програми

Paginator.kt

```
package com.demoss.paginatorrenderkit

import com.demoss.paginatorrenderkit.Paginator.Action
import com.demoss.paginatorrenderkit.Paginator.State
import com.demoss.paginatorrenderkit.Paginator.Store
import com.demoss.paginatorrenderkit.Paginator.reducer
import timber.log.Timber

/**
 * Original idea and huge peace of code is taken from GitFox client source code
 * all links are provided in @see[README.md]
 * or @see[https://github.com/DeMoss15/PaginatorRenderKit]
 *
 * Changes:
 * Added @param[T] for [State], [Action], [Store] and [reducer] for more
restrictive data type workflow
 */
object Paginator {

    /**
     * Changes:
     * Added [getStateData]
     */
    sealed class State<T> {

        open fun getStateData(): List<T> = emptyList<T>()

        class Empty<T> : State<T>()
        class EmptyProgress<T> : State<T>()
        data class EmptyError<T>(val error: Throwable) : State<T>()
        data class Data<T>(val pageCount: Int, val data: List<T>) : State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class Refresh<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class NewPageProgress<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }

        data class FullData<T>(val pageCount: Int, val data: List<T>) :
State<T>() {
            override fun getStateData(): List<T> = data
        }
    }

    /**
     * Changes:
```

```
     * Added action @see[Action.EditCurrentStateData] for compatibility with
realtime paginated list
     * changes (insert, remove, update etc.)
     */
    sealed class Action<T> {
        class Refresh<T> : Action<T>()
        class Restart<T> : Action<T>()
        class LoadMore<T> : Action<T>()
        data class NewPage<T>(val pageNumber: Int, val items: List<T>, val
isLastPage: Boolean) : Action<T>()
        data class PageError<T>(val error: Throwable) : Action<T>()
        data class EditCurrentStateData<T>(val transaction: (data: List<T>) ->
List<T>) : Action<T>()
    }

    /**
     * Changes:
     * Added side effect @see[SideEffect.CancelLoadings] for request
optimization and avoid content
     * duplication
     */
    sealed class SideEffect {
        data class LoadPage(val currentPage: Int) : SideEffect()
        data class ErrorEvent(val error: Throwable) : SideEffect()
        object CancelLoadings : SideEffect()
    }

    /**
     * The method contains logic for changing states @see[State] depending on
     * @see[Action] and current state
     * Also has side effects @see[SideEffect] to inform business layer about the
state's needs
     *
     * Changes:
     * Added side effect @see[SideEffect.CancelLoadings]
     * Added handling of @see[Action.EditCurrentStateData]
     * Fixed @see[SideEffect.LoadPage] for each @see[Action.Refresh]
     *
     * @param T items' type
     * @param state current state stored in @see[Store]
     * @param sideEffectListener informs about state's needs
     */
    private fun <T> reducer(
        action: Action<T>,
        state: State<T>,
        sideEffectListener: (SideEffect) -> Unit
    ): State<T> = when (action) {
        is Action.Refresh -> {
            when (state) {
                is State.Empty -> State.EmptyProgress()
                is State.EmptyError -> State.EmptyProgress()
                is State.Data -> {
                    sideEffectListener(SideEffect.LoadPage(1))
                    State.Refresh(state.pageCount, state.data)
                }
                is State.NewPageProgress -> {
                    sideEffectListener(SideEffect.LoadPage(1))
                    State.Refresh(state.pageCount, state.data)
                }
                is State.FullData -> {
                    sideEffectListener(SideEffect.LoadPage(1))
                    State.Refresh(state.pageCount, state.data)
```

```
            }
            else -> state
        }
    }
    is Action.Restart -> {
        sideEffectListener(SideEffect.CancelLoadings)
        sideEffectListener(SideEffect.LoadPage(1))
        when (state) {
            is State.Empty -> State.EmptyProgress()
            is State.EmptyError -> State.EmptyProgress()
            is State.Data -> State.EmptyProgress()
            is State.Refresh -> State.EmptyProgress()
            is State.NewPageProgress -> State.EmptyProgress()
            is State.FullData -> State.EmptyProgress()
            else -> state
        }
    }
    is Action.LoadMore -> {
        when (state) {
            is State.Data -> {
                sideEffectListener(SideEffect.LoadPage(state.pageCount + 1))
                State.NewPageProgress(state.pageCount, state.data)
            }
            else -> state
        }
    }
    is Action.NewPage -> {
        val items = action.items
        when (state) {
            is State.EmptyProgress -> {
                if (items.isEmpty()) {
                    State.Empty()
                } else {
                    if (action.isLastPage) {
                        State.FullData(action.pageNumber, items)
                    } else {
                        State.Data(action.pageNumber, items)
                    }
                }
            }
            is State.Refresh -> {
                if (items.isEmpty()) {
                    State.Empty()
                } else {
                    if (action.isLastPage) {
                        State.FullData(action.pageNumber, items)
                    } else {
                        State.Data(action.pageNumber, items)
                    }
                }
            }
            is State.NewPageProgress -> {
                if (action.isLastPage) {
                    State.FullData(action.pageNumber, state.data + items)
                } else {
                    State.Data(action.pageNumber, state.data + items)
                }
            }
            else -> state
        }
    }
    is Action.PageError -> {
```

```
            when (state) {
                is State.EmptyProgress -> State.EmptyError(action.error)
                is State.Refresh -> {
                    sideEffectListener(SideEffect.ErrorEvent(action.error))
                    State.Data(state.pageCount, state.data)
                }
                is State.NewPageProgress -> {
                    sideEffectListener(SideEffect.ErrorEvent(action.error))
                    State.Data(state.pageCount, state.data)
                }
                else -> state
            }
        }
        is Action.EditCurrentStateData -> {
            val editedData = action.transaction(state.getStateData())
            if (editedData == state.getStateData()) {
                // nothing changed, return current state
                state
            } else if (editedData.isEmpty()) {
                // edited and state data are different
                // edited is empty
                when(state) {
                    // nothing changes for empty states
                    is State.Empty -> state
                    is State.EmptyProgress -> state
                    is State.EmptyError -> state
                    // states with data change to empty alternative
                    is State.Data -> State.Empty()
                    is State.Refresh -> State.EmptyProgress()
                    is State.NewPageProgress -> State.EmptyProgress()
                    is State.FullData -> State.Empty()
                }
            } else {
                // edited and state data are different
                // edited is not empty
                when (state) {
                    // change empty state to non-empty alternative
                    is State.Empty -> State.FullData(1, editedData)
                    is State.EmptyProgress -> State.NewPageProgress(1,
editedData)
                    is State.EmptyError -> State.FullData(1, editedData)
                    // update data in not-empty states
                    is State.Data -> State.Data(state.pageCount, editedData)
                    is State.Refresh -> State.Refresh(state.pageCount,
editedData)
                    is State.NewPageProgress ->
State.NewPageProgress(state.pageCount, editedData)
                    is State.FullData -> State.FullData(state.pageCount,
editedData)
                }
            }
        }
    }

    /**
     * Store represents Paginator states to business layer
     * Every time you want to use Paginator you should create a new Store
     * Don't forget to set @property[render] and @property[executeSideEffect]
     *
     * Changes:
     * Removed RxJava dependencies
     * Added tag to logs
```

```
     *
     * @param T states' data type
     */
    class Store<T> {

        companion object {
            const val PAGINATOR_LOG_TAG = "PAGINATOR"
        }

        private var state: State<T> = State.Empty()
        var render: (State<T>) -> Unit = {}
            set(value) {
                field = value
                value(state)
            }
        var executeSideEffect: (SideEffect) -> Unit = {}

        fun proceed(action: Action<T>) {
            Timber.tag(PAGINATOR_LOG_TAG).d("Action: $action")
            val newState = reducer(action, state) { sideEffect ->
                executeSideEffect(sideEffect)
            }
            if (newState != state) {
                state = newState
                Timber.tag(PAGINATOR_LOG_TAG).d("New state: $state")
                render(state)
            }
        }
    }
}
```

PaginatorViewDelegate.kt

```
package com.demoss.paginatorrenderkit.view.delegate

import android.view.View
import androidx.recyclerview.widget.RecyclerView
import androidx.swiperefreshlayout.widget.SwipeRefreshLayout
import com.demoss.paginatorrenderkit.Paginator

/**
 * Delegate for setup views in accordance to current @link[Paginator.State]
 *
 * @param refreshCallback triggered by @param[swipeToRefresh]
 * @param adapter will be set to recycler view and notified about new data form
new state @see[render]
 * @param emptyView displays empty data and empty error states
 *
 * @author Daniel Mossur
 */

class PaginatorViewDelegate<T : Any>(
    refreshCallback: (() -> Unit)? = null,
    private val adapter: AbsPaginatorAdapter<in T>,
    private val recyclerView: RecyclerView,
    private val swipeToRefresh: SwipeRefreshLayout,
    private val emptyView: AbsPaginatorEmptyView,
    private val fullscreenProgressView: View
) {
```

```
    /**
     * @param refreshCallback triggered by @param swipeToRefresh
     * @param adapter will be set to recycler view and notified about new data
form new state @see[render]
     * @param paginatorView provides: @property[recyclerView],
@property[swipeToRefresh], @property[emptyView],
@property[fullscreenProgressView]
     * @constructor An alternative to primary constructor
     */
    constructor(
        refreshCallback: (() -> Unit)? = null,
        adapter: AbsPaginatorAdapter<in T>,
        paginatorView: AbsPaginatorView
    ) : this(
        refreshCallback,
        adapter,
        paginatorView.getRecyclerView(),
        paginatorView.getSwipeRefreshLayout(),
        paginatorView.getEmptyView(),
        paginatorView.getFullScreenProgressView()
    )

    init {
        swipeToRefresh.setOnRefreshListener { refreshCallback?.invoke() }
        recyclerView.adapter = adapter
    }

    /**
     * The only and the main purpose of the delegate - setup views depending on
new state
     *
     * @param state could be received from @see[Paginator.Store.render]
     */
    fun render(state: Paginator.State<out T>) {
        recyclerView.post {
            when (state) {
                is Paginator.State.Empty -> {
                    swipeToRefresh.isRefreshing = false
                    fullscreenProgressView.visible(false)
                    adapter.fullData = true
                    adapter.update(emptyList(), false)
                    emptyView.showEmptyData()
                    swipeToRefresh.visible(true)
                }
                is Paginator.State.EmptyProgress -> {
                    swipeToRefresh.isRefreshing = false
                    fullscreenProgressView.visible(true)
                    adapter.fullData = false
                    adapter.update(emptyList(), false)
                    emptyView.visible(false)
                    swipeToRefresh.visible(false)
                }
                is Paginator.State.EmptyError -> {
                    swipeToRefresh.isRefreshing = false
                    fullscreenProgressView.visible(false)
                    adapter.fullData = false
                    adapter.update(emptyList(), false)
                    emptyView.showEmptyError(state.error)
                    swipeToRefresh.visible(true)
                }
                is Paginator.State.Data -> {
                    swipeToRefresh.isRefreshing = false
```

```
                    fullscreenProgressView.visible(false)
                    adapter.fullData = false
                    adapter.update(state.data, false)
                    emptyView.visible(false)
                    swipeToRefresh.visible(true)
                }
                is Paginator.State.Refresh -> {
                    swipeToRefresh.isRefreshing = true
                    fullscreenProgressView.visible(false)
                    adapter.fullData = false
                    adapter.update(state.data, false)
                    emptyView.visible(false)
                    swipeToRefresh.visible(true)
                }
                is Paginator.State.NewPageProgress -> {
                    swipeToRefresh.isRefreshing = false
                    fullscreenProgressView.visible(false)
                    adapter.fullData = false
                    adapter.update(state.data, true)
                    emptyView.visible(false)
                    swipeToRefresh.visible(true)
                }
                is Paginator.State.FullData -> {
                    swipeToRefresh.isRefreshing = false
                    fullscreenProgressView.visible(false)
                    adapter.fullData = true
                    adapter.update(state.data, false)
                    emptyView.visible(false)
                    swipeToRefresh.visible(true)
                }
            }
        }
    }

    private fun View.visible(isVisible: Boolean) {
        visibility = if (isVisible) View.VISIBLE else View.GONE
    }
}
...
```

ДОДАТОК Б

ВІДГУК

керівника економічного розділу

на кваліфікаційну роботу магістра

на тему:

«Оптимізація синхронізації, керування та відтворення станів списків з посторінковим відтворенням»

студента групи 122м-19-1 Моссура Данила Євгенійовича

Керівник економічного розділу
доцент каф. ПЕП та ПУ, к.е.н. **Л. В. Касьяненко**

ДОДАТОК В

Список файлів на диску

Назва файлу	Описання
Пояснювальні документі	
Кваліфікаційна_робота_МоссурДЄ.docx	Пояснювальна записка до кваліфікаційної роботи MS WORD
Кваліфікаційна_робота_МоссурДЄ.pdf	Пояснювальна записка до кваліфікаційної роботи PDF
Програма	
PaginalRenderKit.zip	Архів з вихідним кодом
Презентація	
Кваліфікаційна_робота_МоссурДЄ.ppt	Слайди для презентації кваліфікаційної роботи
Додаткові документи	
PaginalRenderKit.mdj	Діаграми використані у роботі