Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту

Завідувач кафедри

________Олена ОЛЬХОВСЬКА

(підпис)

«___»_________2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на тему

«АЛГОРИТМІЗАЦІЯ ТА РОЗРОБКА НАВЧАЛЬНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З ТЕМИ «ФУНКЦІЙНЕ ПРОГРАМУВАННЯ В JAVA» ДИСЦИПЛІНИ «СУЧАСНІ ПАРАДИГМИ ПРОГРАМУВАННЯ»

зі спеціальності 122 «Комп'ютерні науки»
освітня програма «Комп'ютерні науки»
ступеня бакалавр

Виконавець роботи Рослов Богдан Ігорович

__________«__»______2026 р.

(підпис)

Науковий керівник к.ф.-м.н.,доц., Олексійчук Юрій Федорович

__________«__»______2026 р.

(підпис)

Рецензент

ПОЛТАВА 2026

Завідувач кафедри ________ Олена ОЛЬХОВСЬКА
«10» жовтня 2025 р.

ЗАВДАННЯ І КАЛЕНДАРНИЙ ГРАФІК
ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

на тему «Алгоритмізація та розробка навчального програмного забезпечення з теми «Функційне програмування в Java» дисципліни «Сучасні парадигми програмування»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавр
Прізвище, ім'я, по батькові: Рослов Богдан Ігорович
Затверджена наказом ректора № 213-Н від «01» жовтня 2025 р.
Термін подання студентом роботи «20» травня 2026 р.
Вихідні дані до кваліфікаційної роботи: публікації з теми, загальна постановка задачі, вимоги до програмного продукту.
Зміст пояснювальної записки (перелік питань, які потрібно розробити)

ВСТУП
1. ПОСТАНОВКА ЗАДАЧІ
2. ІНФОРМАЦІЙНИЙ ОГЛЯД
 2.1. Функційне програмування як парадигма
 2.2. Функційне програмування в Java
 2.3 Огляд існуючих навчальних засобів з функційного програмування та їх порівняльний аналіз
3. ТЕОРЕТИЧНА ЧАСТИНА
 3.1. Обґрунтування вибору технологічного стека та архітектурного шаблону
 3.2. Алгоритмізація роботи застосунку
 3.3. Проєктування застосунку
4. ПРАКТИЧНА ЧАСТИНА
 4.1. Реалізація функціоналу застосунку
 4.2. Реалізація підсистеми тренажера та графічного інтерфейсу
 4.3. Тестування програмного продукту
ВИСНОВОК
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ДОДАТОК А

Перелік графічного матеріалу: 3-4 аркуші графічного матеріалу, інші необхідні ілюстрації.

Консультанти розділів кваліфікаційної роботи

Розділ	ПІБ, посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Постанова задачі	Олексійчук Ю.Ф.	15.10.25	15.10.25
Інформаційний огляд	Олексійчук Ю.Ф	15.10.25	15.10.25
Теоретична частина	Олексійчук Ю.Ф.	15.10.25	15.10.25
Практична реалізація	Олексійчук Ю.Ф	15.10.25	15.10.25

Календарний графік виконання кваліфікаційної роботи

Зміст роботи	Термін виконання	Фактичне виконання
1. Вступ	15.05.26	
2. Вивчення методичних рекомендацій і стандартів та звіт керівнику	01.02.26	
3. Постановка завдання	15.02.26	
4. Інформаційний огляд джерел бібліотек та Інтернету	01.03.26	
5. Теоретична частина	01.04.26	
6. Практична частина	01.05.26	
7. Закінчення оформлення	15.05.26	
8. Доповідь студента на кафедрі	20.05.26	
9. Доопрацювання (за необхідності), рецензування	25.05.26	

Дата видачі завдання «15» жовтня 2025 р.

Здобувач вищої освіти Рослов Богдан Ігорович

Науковий керівник к. ф.-м. н. Олексійчук Ю.Ф

Результати захисту кваліфікаційної роботи

Кваліфікаційна робота оцінена на ______________________________

(балів, оцінка за національною шкалою, оцінка за ЄКТС)

Протокол засідання ЕК № ____ від «____» ____________ 2026 р.

Секретар ЕК ______________ ________________________________

«__» _______ 2026 р. «__» _______ 2026 р.

План

кваліфікаційної роботи на тему

«Алгоритмізація та розробка навчального програмного забезпечення з теми «Функційне програмування в Java» дисципліни «Сучасні парадигми програмування» зі спеціальності 122 Комп'ютерні науки освітня програма 122 Комп'ютерні науки ступеня бакалавр

Прізвище, ім'я, по батькові Рослов Богдан Ігорович

ВСТУП

1. ПОСТАНОВКА ЗАДАЧІ
2. ІНФОРМАЦІЙНИЙ ОГЛЯД
   - 2.1. Функційне програмування як парадигма
   - 2.2. Функційне програмування в Java
   - 2.3. Огляд існуючих навчальних засобів з функційного програмування та їх порівняльний аналіз
3. ТЕОРЕТИЧНА ЧАСТИНА
   - 3.1. Обґрунтування вибору технологічного стека та архітектурного шаблону
   - 3.2. Алгоритмізація роботи застосунку
   - 3.3. Проєктування застосунку
4. ПРАКТИЧНА ЧАСТИНА
   - 4.1. Реалізація функціоналу застосунку
   - 4.2. Реалізація підсистеми тренажера та графічного інтерфейсу
   - 4.3. Тестування програмного продукту

ВИСНОВКИ

Здобувач вищої освіти ___________________ Богдан РОСЛОВ «____» __________ 2026 р.

АНОТАЦІЯ

У дипломній роботі розглянуто питання алгоритмізації та розробки навчального програмного забезпечення з теми «Функційне програмування в Java» дисципліни «Сучасні парадигми програмування». Проведено аналіз функційного програмування як сучасної парадигми розробки програмного забезпечення, досліджено особливості реалізації функційного підходу в мові Java та виконано огляд існуючих навчальних засобів для вивчення цієї тематики.

У роботі сформульовано функціональні та нефункціональні вимоги до програмного продукту, обґрунтовано вибір технологічного стека на основі Java 21, JavaFX, SQLite, Maven та JUnit 5. Розроблено архітектуру програмного забезпечення, алгоритми функціонування окремих модулів, структуру бази даних та графічний інтерфейс користувача.

Результатом роботи є настільний навчальний тренажер, який забезпечує інтерактивне вивчення основ функційного програмування в Java, демонстрацію можливостей Stream API, Optional та композиції функцій, а також контроль знань за допомогою системи тестування. Реалізовано механізми збереження профілів користувачів, обліку навчального прогресу, адміністрування банку питань і експорту результатів.

Проведене тестування підтвердило коректність роботи програмного забезпечення та його відповідність поставленим вимогам. Розроблений програмний продукт може бути використаний у навчальному процесі закладів вищої освіти для підвищення ефективності засвоєння теми «Функційне програмування в Java» та організації самостійної роботи студентів.

Ключові слова: функційне програмування, навчальний тренажер, програмне забезпечення, Java, Stream API, Optional, JavaFX, SQLite.

ЗМІСТ

ВСТУП ... 7
РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ ... 10
РОЗДІЛ 2. ІНФОРМАЦІЙНИЙ ОГЛЯД ... 14
2.1. Функційне програмування як парадигма ... 14
2.2. Функційне програмування в Java ... 17
2.3. Огляд існуючих навчальних засобів з функційного програмування та їх порівняльний аналіз ... 20
РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА ... 23
3.1. Обґрунтування вибору технологічного стека та архітектурного шаблону ... 23
3.2. Алгоритмізація роботи застосунку ... 29
3.3. Проєктування застосунку ... 40
РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА ... 43
4.1. Реалізація функціоналу застосунку ... 43
4.2. Реалізація графічного інтерфейсу ... 51
4.3. Тестування програмного продукту ... 62
ВИСНОВКИ ... 69
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ ... 71
ДОДАТКИ ... 77

ВСТУП

Сучасна інженерія програмного забезпечення характеризується мультипарадигмальністю: поряд із об'єктно-орієнтованим підходом промислові мови активно впроваджують засоби функційного програмування. У Java функційні можливості – лямбда-вирази, Stream API, Optional, композиція через andThen/compose, колектори Collectors – стали невід'ємною частиною платформи починаючи з версії 8 (2014 рік) і продовжують розвиватися у версіях 17–21 LTS. Знання цих засобів є базовою кваліфікаційною вимогою для розробників, що підтверджується їх присутністю в робочих програмах дисциплін «Сучасні парадигми програмування», «Алгоритмізація та програмування» профільних спеціальностей.

Аналіз навчального процесу засвідчує, що засвоєння функційних засобів Java пов'язане з характерними утрудненнями: студенти плутають проміжні та термінальні операції стрімів, помилково використовують orElse замість orElseGet, неправильно застосовують вкладені колектори. Подолання цих утруднень потребує систематичної роботи з інтерактивними прикладами та значної кількості практичних задач з негайним зворотним зв'язком, що актуалізує потребу у спеціалізованому навчальному тренажері.

Огляд існуючих засобів – інтерактивних платформ (Codecademy, HyperSkill, Exercism), документації (Oracle Tutorials, Baeldung), інструментів IDE – показав відсутність локального настільного застосунку, який одночасно надає інтерактивні модулі для експериментування з концепціями ФП, банк питань для контролю знань, журнал індивідуального прогресу та теоретичну довідку, працюючи без підключення до Інтернету і без реєстрації.

Актуальність теми зумовлена необхідністю підвищення якості навчального процесу з теми «Функційне програмування в Java» шляхом впровадження інтерактивного програмного засобу, що поєднує демонстрацію концепцій, практику кодування та контроль засвоєння.

Об'єкт дослідження – процес навчання здобувачів принципам і засобам функційного програмування в мові Java у межах дисципліни «Сучасні парадигми програмування».

Предмет дослідження – методи алгоритмізації навчальних сценаріїв і програмні засоби реалізації навчального тренажера з функційного програмування в Java.

Мета роботи – спроєктувати та програмно реалізувати настільний навчальний тренажер з теми «Функційне програмування в Java», який забезпечує інтерактивну демонстрацію ключових засобів ФП, контроль знань через банк питань трьох типів, фіксацію індивідуального прогресу та теоретичну довідку.

Для досягнення поставленої мети сформульовано такі задачі:

– проаналізувати функційне програмування як парадигму та визначити ключові концепції, що мають бути відображені у тренажері (чисті функції, незмінність, функції вищого порядку, композиція, декларативність, Optional);
– виконати огляд існуючих навчальних засобів з ФП у Java та обґрунтувати потребу в розробці нового продукту;
– сформулювати функціональні та нефункціональні вимоги до тренажера;
– обґрунтувати вибір технологічного стека (Java 21, JavaFX 26, SQLite, Maven, JUnit 5) та спроєктувати трирівневу архітектуру;
– алгоритмізувати навчальні сценарії та сценарій контролю знань;
– спроєктувати реляційну модель даних (6 таблиць) та графічний інтерфейс (2 екрани, 8 вкладок);
– програмно реалізувати чотири демонстраційні модулі (Stream API, аналіз тексту, конвеєр функцій, Optional) та підсистему тренажера з трьома типами питань;
– провести тестування на трьох рівнях (26 модульних тестів JUnit 5, 14 ручних перевірок, верифікація 18 вимог).

Методи дослідження: аналіз літературних джерел та технічної документації, порівняльний аналіз існуючих рішень, об'єктно-орієнтоване проєктування з елементами функційного підходу, модульне та функціональне тестування.

Практичне значення. Розроблений тренажер може використовуватися у навчальному процесі для самостійної роботи студентів, підготовки до практичних занять та модульного контролю з теми «Функційне програмування в Java». Банк із 32 питань п'яти тем покриває основні засоби ФП у Java; адмін-вкладка дозволяє викладачеві розширювати банк без перекомпіляції.

Структура роботи. Робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та додатків. У розділі 1 проаналізовано предметну область, здійснено огляд існуючих засобів та сформульовано вимоги. У розділі 2 описано проєктні рішення: стек, архітектура, алгоритми, модель БД, GUI. У розділі 3 подано реалізацію функціональних модулів, тренажера, інтерфейсу та результати тестування. Загальний обсяг програмного коду – 3 522 рядки Java (33 файли), 611 рядків FXML (10 файлів), 187 рядків CSS.

РОЗДІЛ 1. ПОСТАНОВКА ЗАДАЧІ

На основі попереднього аналізу сформульовано задачу бакалаврської роботи: спроєктувати та програмно реалізувати настільний навчальний тренажер з теми «Функційне програмування в Java» дисципліни «Сучасні парадигми програмування», який забезпечує інтерактивну демонстрацію ключових засобів ФП, контроль рівня засвоєння через банк питань, фіксацію індивідуального прогресу студента та теоретичну довідку. У підрозділі формалізуються функціональні та нефункціональні вимоги до програмного продукту.

Функціональні вимоги визначають, що саме повинна робити система. Перелічимо їх з обґрунтуванням.

– Ф1. Інтерактивна демонстрація операцій Stream API на колекції даних. Користувач має змогу переглядати таблицю студентів, додавати, видаляти записи, виконувати типові операції (фільтрування за порогом, сортування за середнім балом, групування за містом і навчальною групою, обчислення середнього, підрахунок кількості елементів за групами, агрегована статистика по всіх оцінках). Для кожної операції система відображає не лише результат, а й фрагмент коду, який було еквівалентно виконано, що сприяє розумінню зв'язку між інтерфейсом і кодом.
– Ф2. Аналіз тексту засобами Stream API. Користувач уводить довільний текст; система здійснює токенізацію, обчислює частотний словник, відображає топ-N найчастіших слів у вигляді таблиці та гістограми; виводить підсумкові показники – загальну кількість слів, кількість унікальних слів, середню довжину, найдовше слово.
– Ф3. Конструктор конвеєра обробки чисел. Користувач формує послідовність простих перетворень (множення, додавання, фільтр за умовою, модуль, заперечення, піднесення у квадрат) із параметрами; система комбінує їх через композицію функцій (Function.andThen) і

застосовує до вхідного списку чисел, виводячи результат і агреговану статистику.

- Ф4. Демонстрація патернів роботи з Optional. Чотири незалежні модулі: безпечний парсинг цілого числа з рядка, безпечне ділення з контролем нуля, безпечне отримання довжини рядка та верхнього регістру, лінива підстановка через orElseGet.
- Ф5. Тренажер контролю знань. Питання зберігаються у локальній базі даних. Підтримуються три типи питань: одиничний вибір, множинний вибір, заповнення (FILL_IN). Перед початком тренування користувач обирає теми та кількість питань. Питання та варіанти відповідей перемішуються випадковим чином, щоб запам'ятовування «правильна відповідь – другий варіант» не давало переваги.
- Ф6. Профілі користувачів. Кожен студент має власний профіль (прізвище та ім'я, навчальна група). Журнал спроб ведеться окремо для кожного профілю.
- Ф7. Облік прогресу. Для кожного користувача обчислюються зведені показники: загальна кількість завершених спроб, середній бал, найкращий результат, останній результат. Журнал спроб подається у вигляді таблиці з датами; динаміка результатів візуалізується лінійним графіком.
- Ф8. Теоретичний довідник. У застосунку реалізовано вкладку з теоретичними матеріалами по кожній темі, що дає змогу студентові освіжити знання перед тренуванням.
- Ф9. Експорт результатів у CSV-файл, який може бути відкритий у будь-якій програмі електронних таблиць. Сама операція експорту реалізована через Stream API і Collectors.joining – таким чином, цей експорт сам є демонстрацією одного з засобів ФП.
- Ф10. Адміністрування банку питань. Викладач (або сам студент у режимі експерименту) має змогу додавати, редагувати, видаляти питання та варіанти відповідей без перекомпіляції програми.

Нефункціональні вимоги визначають характеристики продукту, які впливають на його якість, надійність та зручність експлуатації.

– НФ1. Зручність використання. Інтерфейс має бути інтуїтивно зрозумілим, спиратися на стандартні елементи керування, не вимагати від користувача читання інструкції. Усі основні операції доступні з вкладок, групування елементів логічне і відповідає сценаріям роботи.
– НФ2. Локальність. Програма повинна працювати без постійного з'єднання з Інтернетом. Усі дані (банк питань, профілі користувачів, журнал спроб) зберігаються локально у файлі бази даних.
– НФ3. Кросплатформенність. Застосунок має запускатися на всіх основних настільних операційних системах (Windows, Linux, macOS) без зміни вихідного коду.
– НФ4. Продуктивність. Час відгуку на типові операції користувача (виконання Stream-операції на демо-колекції, перевірка відповіді, перехід до наступного питання) не повинен перевищувати 200 мс.
– НФ5. Надійність. Помилки введення (некоректні числа, порожні поля, дублікати) обробляються коректно, з виведенням зрозумілих повідомлень. Збій бази даних або файлу налаштувань не повинен призводити до аварійного завершення.
– НФ6. Розширюваність. Архітектура має бути такою, щоб додавання нової теми або типу питання не вимагало переписування великих частин коду. Новий тип питання додається через розширення переліку Question.Type і відповідної гілки в QuizService.
– НФ7. Тестованість. Чиста функційна логіка (модулі StudentsAnalytics, TextAnalytics, PipelineBuilder, OptionalDemo, QuizService) має бути покрита автоматичними модульними тестами.
– НФ8. Відповідність парадигмі. Оскільки тренажер сам присвячено функційному програмуванню, його внутрішня реалізація демонстраційних модулів має активно використовувати засоби ФП, щоб бути зразковою як з освітнього, так і з технічного боку.

Технологічні обмеження. Реалізація має бути виконана мовою Java версії не нижчої за Java 17 LTS. Вибір саме Java обумовлений предметом дисципліни – тренажер навчає функційного програмування саме у Java. Графічний інтерфейс будується засобами JavaFX як офіційного фреймворку графічних застосунків Java SE. Для збереження даних використовується вбудована реляційна СУБД SQLite, яка не потребує окремого серверу і зберігає всю базу в одному файлі. Для управління збиранням і залежностями використовується Apache Maven; для модульного тестування – JUnit 5; як середовище розробки – Microsoft Visual Studio Code з пакетом розширень Extension Pack for Java [10].

Цільова аудиторія застосунку – студенти 2–3 курсів спеціальностей у галузі знань «Інформаційні технології», які вивчають дисципліну «Сучасні парадигми програмування» або суміжні. Ризики проєкту, пов'язані з відсутністю мережевого доступу, відсутністю серверу, потребою мати лише інтерпретатор JVM, повністю усуваються обраним технологічним стеком.

Критерії приймання роботи. Розроблений програмний продукт вважається таким, що відповідає поставленій задачі, якщо: (а) реалізовано всі десять функціональних вимог Ф1–Ф10; (б) виконуються всі нефункціональні вимоги НФ1–НФ8; (в) код програми компілюється без помилок і запускається на еталонній конфігурації (Java 17+, JavaFX 21+, SQLite 3.46+); (г) усі автоматичні модульні тести проходять успішно; (д) візуально перевірено сценарії проходження тренування на банку з не менш ніж 30 питаннями за п'ятьма темами курсу.

РОЗДІЛ 2. ІНФОРМАЦІЙНИЙ ОГЛЯД

2.1. Функційне програмування як парадигма

Парадигма програмування – це сукупність ідей і способів мислення, які визначають спосіб формулювання обчислень. У сучасній програмній інженерії співіснують декілька основних парадигм: імперативна, об'єктно-орієнтована, функційна, логічна та конкурентна. Кожна з них пропонує власну модель опису задачі та власні засоби побудови програмних рішень. Сучасні мови загального призначення (Java, C#, Python, Kotlin, Scala) є мультипарадигмальними – вони об'єднують засоби кількох парадигм у межах однієї мови, що дозволяє обирати стиль розв'язання задачі залежно від її природи [1].

Функційне програмування (далі – ФП) – це парадигма, у якій обчислення формулюється як обчислення значень функцій, а не як послідовність операцій над станом. У чистому функційному стилі програма складається з функцій, які приймають аргументи і повертають результат, не змінюючи зовнішніх змінних і не маючи прихованих побічних ефектів. Такі функції називають чистими (pure) – їх результат однозначно визначається аргументами, що робить поведінку програми передбачуваною та полегшує її тестування [2].

Історично функційний підхід ґрунтується на λ-обчисленні А. Чорча, сформульованому ще у 1930-х роках як математична формалізація поняття обчислюваності. Перші програмні втілення цієї ідеї з'явилися наприкінці 1950-х років у мові LISP, розробленій Дж. Маккарті. Подальший розвиток парадигми пов'язаний з мовами ML, Haskell, Erlang, OCaml, Scheme, F#, Clojure. У XXI столітті функційні засоби почали активно вбудовуватися у мейнстрімні промислові мови. У Java функційні риси з'явилися у версії 8 (2014 рік) – це лямбда-вирази, функціональні інтерфейси, посилання на методи, Stream API, клас Optional [3].

Ключові ознаки функційного програмування, що відрізняють його від імперативного підходу, такі.

– Функції як значення першого класу. Функцію можна передати як аргумент, повернути з іншої функції, зберегти у змінній або колекції. Завдяки цьому стають можливими шаблони вищого порядку: composeFunctions, mapReduce, filterCollect.
– Незмінність даних (immutability). Структура даних після створення не змінюється; для отримання модифікованого значення створюється нова копія. Незмінні структури безпечні для конкурентного використання та простіші у відлагодженні.
– Чисті функції та контроль побічних ефектів. Чиста функція не модифікує зовнішній стан, не виконує введення-виведення, не залежить від глобальних змінних. Це забезпечує референційну прозорість – виклик функції можна замінити її значенням без зміни поведінки програми.
– Декларативний стиль. Програміст описує, що потрібно отримати, а не покрокову послідовність дій. Цикли замінюються операціями над колекціями (map, filter, reduce), розгалуження – чистими функціями.
– Композиція функцій. Складні перетворення формуються з простих за допомогою явних операцій композиції (andThen, compose), що сприяє повторному використанню коду.

Згідно з підходом, описаним у роботах Б. Гетца [4], для практики промислової розробки чистий функційний стиль рідко використовують у відриві від інших парадигм. Натомість продуктивним виявляється поєднання функційних засобів з об'єктно-орієнтованим декомпозиційним підходом: функційний стиль застосовується до обробки даних та алгоритмічних перетворень, а об'єктно-орієнтований – до структурування коду на класи та модулі.

Дисципліна «Сучасні парадигми програмування», яка викладається студентам спеціальностей «Комп'ютерні науки», «Інженерія програмного забезпечення», «Комп'ютерна інженерія», має на меті ознайомлення майбутніх фахівців з основними сучасними підходами до розробки. Поряд із розглядом

об'єктно-орієнтованої, конкурентної та реактивної парадигм у дисципліні значна увага приділяється функційному програмуванню. Це обумовлено двома чинниками. По-перше, функційні засоби увійшли практично в усі сучасні промислові мови, тому їх знання є базовою кваліфікаційною вимогою. По-друге, функційний стиль ефективно поєднується з паралельною та конкурентною обробкою даних, що є критичним у сучасних розподілених системах [5].

У межах цієї дисципліни тема «Функційне програмування в Java» зазвичай розглядається після опанування основ об'єктно-орієнтованого програмування. Її засвоєння передбачає вивчення лямбда-виразів, посилань на методи, функціональних інтерфейсів пакета java.util.function, Stream API, класу Optional, технік композиції функцій та збирання результатів за допомогою Collectors. Особливість теми полягає у необхідності якісного синтезу теоретичних понять з практикою кодування – студент повинен не лише знати визначення термінів, а й уміти застосовувати відповідні засоби для розв'язання типових задач.

Аналіз навчального процесу засвідчує, що традиційні форми викладання (лекція, практичне заняття, самостійна робота з підручником) не завжди забезпечують достатній рівень засвоєння теми. Студенти часто плутають проміжні та термінальні операції стрімів, помилково використовують метод orElse замість orElseGet, неправильно застосовують groupingBy у поєднанні з вкладеними колекторами. Подолання цих утруднень потребує систематичної роботи з прикладами та значної кількості практичних задач, що, своєю чергою, актуалізує потребу у спеціалізованому навчальному програмному забезпеченні – навчальному тренажері.

2.2. Функційне програмування в Java

Реалізація функційних засобів у Java є, з одного боку, обмеженою порівняно з мовами на кшталт Haskell або Scala (відсутні алгебраїчні типи даних у класичному вигляді, відсутня вбудована лінива обчислювана модель), а з іншого – цілком придатною для промислового використання. У підрозділі розглядаються ключові концепції ФП та засоби, якими вони втілені у платформі Java SE 8 і пізніших версіях.

Лямбда-вирази – стиснутий запис анонімної функції, який реалізує функціональний інтерфейс. Функціональним вважається інтерфейс, що містить рівно один абстрактний метод (single abstract method, SAM). Стандартні функціональні інтерфейси зосереджені у пакеті java.util.function і охоплюють найпоширеніші сигнатури [6].

- Function<T, R> – перетворення T → R, метод apply.
- Predicate<T> – перевірка умови T → boolean, метод test.
- Consumer<T> – дія над значенням T → void, метод accept.
- Supplier<T> – постачальник значень () → T, метод get.
- BinaryOperator<T> – бінарна операція над значеннями одного типу, метод apply.
- UnaryOperator<T>, BiFunction<T, U, R>, BiPredicate<T, U> – узагальнення для двох аргументів та операцій над одним типом.

Окремо існують спеціалізовані варіанти для примітивних типів (IntFunction, LongPredicate, DoubleSupplier тощо), які дозволяють уникнути зайвого автобоксингу та підвищити продуктивність. Лямбда-вираз може захоплювати локальні змінні з оточення, але такі змінні мають бути ефективно фінальними (effectively final): після ініціалізації їх значення не повинно змінюватись. Це обмеження є наслідком моделі замикань Java і запобігає колізіям при паралельному використанні.

Посилання на методи (method references) – синтаксичний цукор для лямбди, яка лише делегує виклик до іншого методу. У Java передбачено чотири форми посилань: на статичний метод (Integer::parseInt), на метод об'єкта (System.out::println), на метод довільного об'єкта типу (String::length), на конструктор (ArrayList::new). Використання посилань на методи робить код лаконічнішим та виразнішим, особливо в ланцюжках перетворень над колекціями.

Stream API – найпотужніший засіб функційного стилю в Java. Stream<T> – це послідовність елементів, з якою можна виконувати композиційні операції. Операції поділяються на проміжні (intermediate) та термінальні (terminal). Проміжні (filter, map, sorted, distinct, peek, limit, skip) повертають новий стрім та виконуються ліниво – фактичне обчислення розпочинається лише після виклику термінальної операції (collect, reduce, count, forEach, min, max, findFirst, anyMatch). Така архітектура дозволяє виконати оптимізації на кшталт об'єднання операцій, ранньої зупинки та паралелізації.

Стрім є одноразовим – після термінальної операції повторне використання неможливе, потрібно отримати новий стрім із джерела. Це обмеження покликане зробити поведінку передбачуваною: стрім не накопичує стану. Прикладом типового конвеєра є фільтрація колекції з наступним групуванням і агрегацією:

```
Map<String, Long> byCity = students.stream()
    .filter(s -> s.averageGrade() >= 80)
    .collect(Collectors.groupingBy(
        Student::getCity, Collectors.counting()));
```

Клас Optional<T> впроваджує безпечну роботу з потенційно відсутніми значеннями. Після запровадження у Java 8 Optional став альтернативою повертанню null із методів, що повинні семантично означати «значення може не існувати». Optional є контейнером, який або містить значення, або порожній. Працювати з ним рекомендується у функційному стилі: через map (перетворення значення, якщо присутнє), flatMap (для функцій, що самі повертають Optional), filter, ifPresent, orElse, orElseGet, orElseThrow [7].

Принципова відмінність orElse від orElseGet полягає у моменті обчислення альтернативного значення. orElse приймає готове значення і обчислює його завжди (стратегія «жадібна»), тоді як orElseGet приймає Supplier і викликає його лише за умови, що Optional порожній (стратегія «лінива»). Якщо альтернативне значення обчислюється дорого або має побічний ефект, рекомендується використовувати саме orElseGet.

Композиція функцій – одна з фундаментальних ідей ФП – реалізована методами Function.andThen та Function.compose. Виклик f.andThen(g) повертає нову функцію, яка спочатку застосовує f, а потім g; виклик f.compose(g) – навпаки, спочатку g, потім f. Зазначимо, що Function.identity() повертає нейтральний елемент композиції – функцію x → x. Його зручно використовувати як стартове значення в редукції під час складання конвеєра з колекції функцій.

Колектори (Collectors) – статичні фабричні методи з пакета java.util.stream, що інкапсулюють шаблони збирання результату стріму. Найуживаніші колектори – toList, toSet, toMap, joining, groupingBy, partitioningBy, counting, summingInt, averagingDouble, mapping, reducing. Колектори можна вкладати: другий аргумент groupingBy сам є колектором, що дозволяє формувати дворівневі та більш глибокі групування. Завдяки цій композиційності лаконічно виражаються типові аналітичні запити, які в імперативному стилі потребували б кількох циклів і допоміжних змінних [8].

Поряд із зазначеними засобами Java підтримує паралельні стріми (parallelStream), які автоматично розподіляють обчислення між робочими потоками з пулу ForkJoinPool. Паралельний стрім ефективний на великих обсягах даних та обчислювально-важких операціях за умови дотримання вимог до асоціативності операцій редукції. Розгляд паралельних стрімів виходить за межі базового рівня вивчення теми, проте їх існування слід враховувати при формуванні навчальних матеріалів – студент має чітко розуміти, що операції в стрімі мають бути не лише чистими, а й безпечними з точки зору паралельного виконання.

Підсумовуючи розглянуті концепції, варто відзначити: ФП у Java – це не окрема ізольована частина мови, а набір засобів, що тісно інтегровані з об'єктно-орієнтованим ядром. Їхнє якісне опанування потребує не запам'ятовування окремих прикладів, а формування цілісного розуміння того, як перейти від імперативного мислення до декларативного. Саме це завдання й покликане вирішувати спеціалізоване навчальне програмне забезпечення.

2.3. Огляд існуючих навчальних засобів з функційного програмування та їх порівняльний аналіз

Існуючі засоби, які можуть бути використані для навчання теми «Функційне програмування в Java», можна умовно поділити на чотири групи: інтерактивні онлайн-платформи з курсами та задачами, документація і туторіали, інтегровані у IDE інструменти підказок, спеціалізовані навчальні застосунки.

До першої групи належать платформи Codecademy, Coursera, edX, freeCodeCamp, HyperSkill, Exercism. Вони пропонують структуровані курси з теоретичним матеріалом, прикладами коду, інтерактивними завданнями та автоматичною перевіркою. Перевагою таких платформ є системний підхід – матеріал поданий у логічній послідовності, а виконання завдань підкріплюється негайним зворотним зв'язком. Недоліком є те, що більшість якісних курсів є платними або потребують реєстрації, а безкоштовні курси часто охоплюють тему ФП у Java лише поверхово, без занурення у Stream API та колектори. Крім того, ці платформи не адаптовані під конкретну навчальну програму вищого навчального закладу і не дозволяють викладачеві контролювати індивідуальний прогрес студентів.

До другої групи належать офіційна документація Oracle [3], навчальний цикл Java Tutorials, статті на Baeldung [9], інтерактивні матеріали Mozilla Developer Network та провідних університетів. Це авторитетні і повні джерела, проте вони мають довідниковий характер: студент знаходить там відповіді на

конкретні питання, але не отримує цілісної навчальної траєкторії з контролем рівня засвоєння.

Третя група – це інструменти підказок усередині IDE: автодоповнення IntelliJ IDEA та Eclipse, статичні аналізатори SonarLint, ErrorProne, SpotBugs. Вони допомагають програмістові уникнути типових помилок під час кодування, але не призначені для навчання як такого – вони ефективні тоді, коли студент уже володіє основами і виправляє конкретні недоліки своєї реалізації.

Четверта група – спеціалізовані навчальні застосунки – є найменшою. Серед існуючих рішень можна виокремити такі.

- Java Visualizer (вебсайт pythontutor.com у режимі Java) – візуалізатор виконання Java-коду, корисний для розуміння станів змінних, але не спеціалізований на ФП.
- Mooc.fi – платформа кафедри інформатики Гельсінського університету, яка пропонує безкоштовні курси з Java; містить розділ про ФП. Орієнтований на самостійне опанування, контроль прогресу можливий лише в межах платформи.
- Окремі open-source репозиторії на платформі GitHub з прикладами та задачами на ФП (наприклад, awesome-java-stream-api). Вони не є повноцінними застосунками – радше зібранням прикладів.

Аналіз перелічених рішень дозволяє виявити прогалину: відсутній невеликий локальний навчальний застосунок, який би одночасно (1) надавав інтерактивні модулі для експериментування із засобами ФП, (2) мав банк навчальних питань, що систематично охоплює тему згідно з робочою програмою дисципліни, (3) фіксував індивідуальний прогрес студента, (4) працював без постійного підключення до Інтернету і без реєстрації. Така прогалина і визначає актуальність розробки навчального тренажера, який є об'єктом цієї бакалаврської роботи.

Для систематизації наведемо порівняльну характеристику зазначених типів засобів за такими критеріями: систематичність матеріалу, наявність контролю

знань, локальна робота без Інтернету, безкоштовність, можливість адаптації під конкретну навчальну дисципліну [15].

Інтерактивні платформи мають високу систематичність та якісний контроль знань, частково безкоштовні, потребують Інтернету і не адаптуються під дисципліну ВНЗ. Документація і туторіали мають середню систематичність, не мають контролю знань, потребують Інтернету, безкоштовні, не адаптуються. Інструменти IDE не мають ні систематичності, ні контролю – це засоби допомоги при кодуванні, не навчанні. Спеціалізовані застосунки нечисленні: одиниці з них мають всі перераховані якості одночасно. Розроблюваний у бакалаврській роботі тренажер позиціонується саме як рішення цієї прогалини – локальний навчальний застосунок з банком питань, журналом прогресу та інтерактивними модулями, який повністю відповідає програмі курсу.

Висновки до розділу. У розділі проаналізовано предметну область – функційне програмування як парадигму та її реалізацію в мові Java. Розглянуто ключові концепції ФП (чисті функції, незмінність, функції вищого порядку, композиція, декларативність) та засоби їх втілення у Java SE 8 і пізніших версіях (лямбда-вирази, функціональні інтерфейси, Stream API, Optional, Collectors). Визначено місце теми у дисципліні «Сучасні парадигми програмування» та зафіксовано основні утруднення, з якими стикаються студенти при її засвоєнні. Здійснено огляд існуючих навчальних засобів і виявлено прогалину – відсутність локального навчального тренажера, що системно охоплює тему ФП у Java з урахуванням потреб навчального процесу ВНЗ. Сформульовано задачу роботи, перелік функціональних і нефункціональних вимог, технологічні обмеження та критерії приймання. У наступному розділі ці вимоги слугуватимуть основою для проєктних рішень.

РОЗДІЛ 3. ТЕОРЕТИЧНА ЧАСТИНА

3.1. Обґрунтування вибору технологічного стека та архітектурного шаблону

Вибір технологічного стека для навчального тренажера визначається двома групами чинників: освітніми та інженерними. З освітньої точки зору, тренажер призначений для вивчення функційного програмування саме в мові Java, тому й сам повинен бути реалізований цією мовою – студент, вивчаючи вихідний код продукту, отримує зразковий приклад застосування тих самих засобів ФП, які він опановує. З інженерної точки зору, стек повинен забезпечувати кросплатформенність, локальність (незалежність від Інтернету), простоту розгортання та мінімум зовнішніх залежностей [11].

У таблиці 3.1 наведено порівняльну характеристику трьох альтернативних стеків, які було розглянуто на етапі попереднього аналізу.

Таблиця 3.1 – Порівняння альтернативних технологічних стеків

Критерій	**Java + JavaFX + SQLite**	**Java + Swing + HSQLDB**	**Electron + React + SQLite**
Мова реалізації	Java 17+	Java 17+	JavaScript / TypeScript
GUI-фреймворк	JavaFX (FXML + CSS)	Swing (застарілий)	HTML + CSS + React
Відповідність темі ФП	Повна	Повна	Часткова (інша мова)
СУБД	SQLite (один файл)	HSQLDB (server mode)	SQLite (better-sqlite3)

Розмір дистрибутива	~25 МБ (fat JAR)	~20 МБ	~120 МБ (Electron)
Кросплатформенність	Windows, Linux, macOS	Windows, Linux, macOS	Windows, Linux, macOS

Продовження таблиці 3.1

Сучасність GUI	Висока (CSS-стилі)	Низька (AWT Look&Feel)	Висока
Складність розгортання	Мінімальна (JRE + JAR)	Мінімальна	Потребує Node.js

Стек на основі Electron + React відкинуто через невідповідність освітній меті: тренажер навчає ФП у Java, а не в JavaScript. Стек Java + Swing є технічно придатним, проте фреймворк Swing вважається застарілим, офіційно не розвивається компанією Oracle з 2012 року та не підтримує сучасні можливості стилізації через CSS. Стек Java + JavaFX + SQLite є оптимальним: він забезпечує повну відповідність мові-предмету, сучасний GUI з FXML-розміткою та CSS-стилізацією, легку вбудовану СУБД без окремого серверу, мінімальний розмір дистрибутива [10].

Мова програмування – Java 21 LTS. Обрано довгостроково підтримувану версію (Long-Term Support), що гарантує стабільність середовища виконання. Java 21 включає всі необхідні засоби ФП (лямбда-вирази, Stream API, Optional, записи – records), а також сучасний синтаксис – текстові блоки, патерн-матчінг, switch-вирази [3].

JavaFX 26 – фреймворк настільних графічних інтерфейсів для Java. Побудований за шаблоном MVC: розмітка інтерфейсу описується у FXML (XML-подібна мова), стилі – у CSS, логіка обробки подій – у Java-контролерах. Модулі javafx-controls і javafx-fxml забезпечують стандартні елементи керування

(TableView, BarChart, LineChart, TabPane, Spinner) та завантаження інтерфейсу з FXML-файлів [10].

SQLite 3.46 – легка реляційна СУБД, яка зберігає всю базу в одному файлі. Не потребує окремого серверного процесу. Підключення здійснюється через JDBC-драйвер xerial sqlite-jdbc. Для навчального застосунку з обсягом даних до кількох мегабайт SQLite є оптимальним вибором: нуль-конфігурація, транзакційність, підтримка SQL-92, зовнішні ключі через PRAGMA foreign_keys [11].

Apache Maven 3.8+ – система управління залежностями та складанням. Конфігурація збирання описана в одному файлі pom.xml. Maven автоматично завантажує бібліотеки з Maven Central, компілює код, запускає тести, формує fat-JAR (shaded JAR) для розповсюдження.

JUnit 5 – фреймворк модульного тестування. Використовується для покриття тестами чистої функційної логіки: StudentsAnalytics, TextAnalytics, PipelineBuilder, OptionalDemo, QuizService. Підтримує параметризовані тести, вкладені класи, розширення [12].

Архітектурний шаблон. Для структурування коду обрано трирівневу архітектуру (three-tier architecture) з чітким розмежуванням відповідальностей між шарами. Виокремлено п'ять шарів, зображених на рисунку 2.1.

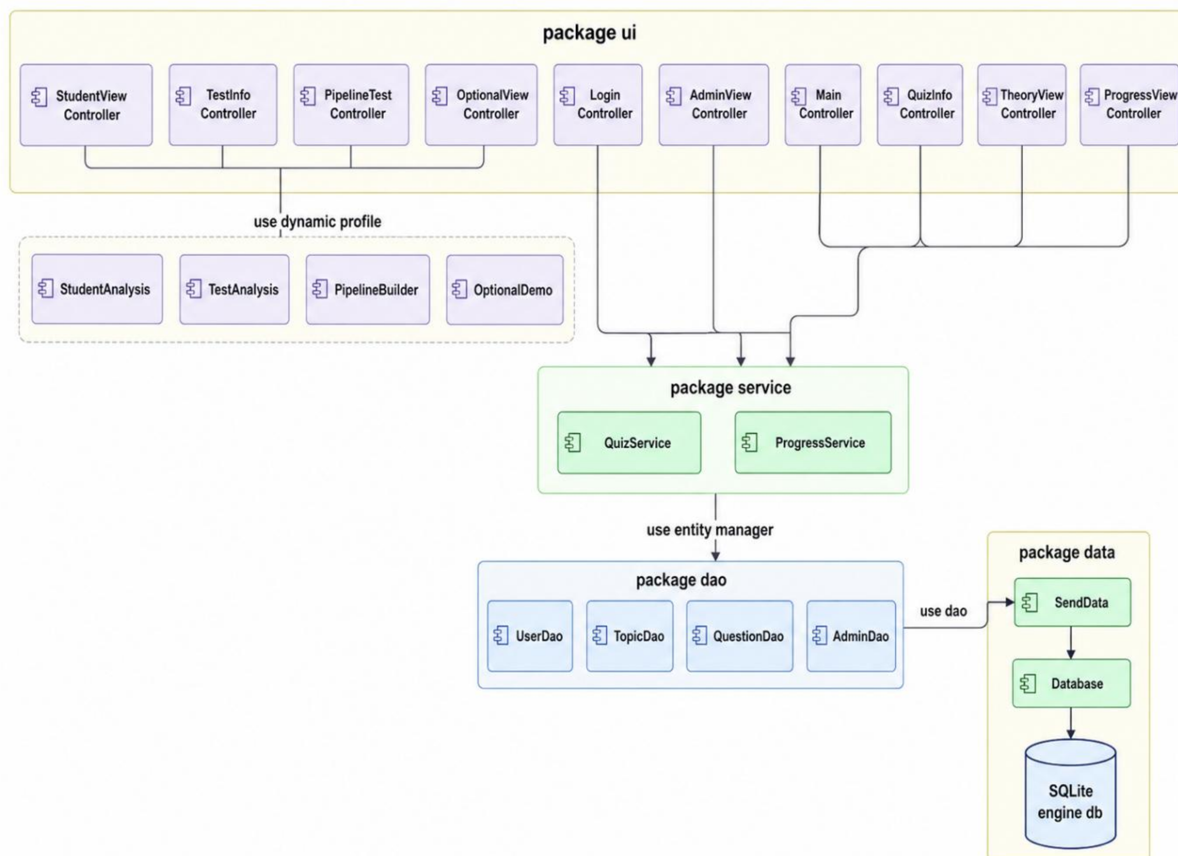

Рисунок 3.1 – Архітектура навчального тренажера

Шар представлення (пакет ui) містить 10 JavaFX-контролерів, кожен з яких обслуговує окрему FXML-розмітку. Контролери не виконують бізнес-логіку та прямих SQL-запитів – вони делегують обчислення відповідним сервісам або функційним модулям.

Шар бізнес-логіки (пакет service) містить три сервіси: QuizService – логіка тренажера (формування набору питань, перевірка відповідей, фіксація спроб), ProgressService – обчислення статистики прогресу, ExportService – генерація CSV-файлів.

Шар функційної логіки (пакет functional) – чисті функції без залежності від БД та GUI. StudentsAnalytics демонструє Stream API на колекції студентів, TextAnalytics – аналіз тексту, PipelineBuilder – композицію функцій, OptionalDemo – патерни Optional. Цей шар ізольований: його класи можна тестувати без запуску JavaFX і без бази даних [13].

Шар доступу до даних (пакет dao) інкапсулює SQL-запити. Кожна DAO-сутність (UserDao, TopicDao, QuestionDao, AttemptDao) виконує CRUD-операції над відповідною таблицею та повертає об'єкти моделі. Методи DAO використовують try-with-resources для гарантованого закриття з'єднань.

Шар даних (пакет db) містить клас Database (ініціалізація схеми, пул з'єднань) та SeedData (заливання початкових даних – тем і банку питань). SQLite-файл створюється автоматично у каталозі користувача при першому запуску [14].

Розподіл класів за пакетами ілюструє діаграма пакетів (Рисунок 2.2). Стрілки показують напрям залежностей: шар представлення залежить від сервісів і функційного шару, сервіси – від DAO і моделі, DAO – від Database і моделі. Зворотних залежностей немає, що забезпечує можливість автономного тестування кожного шару [14].

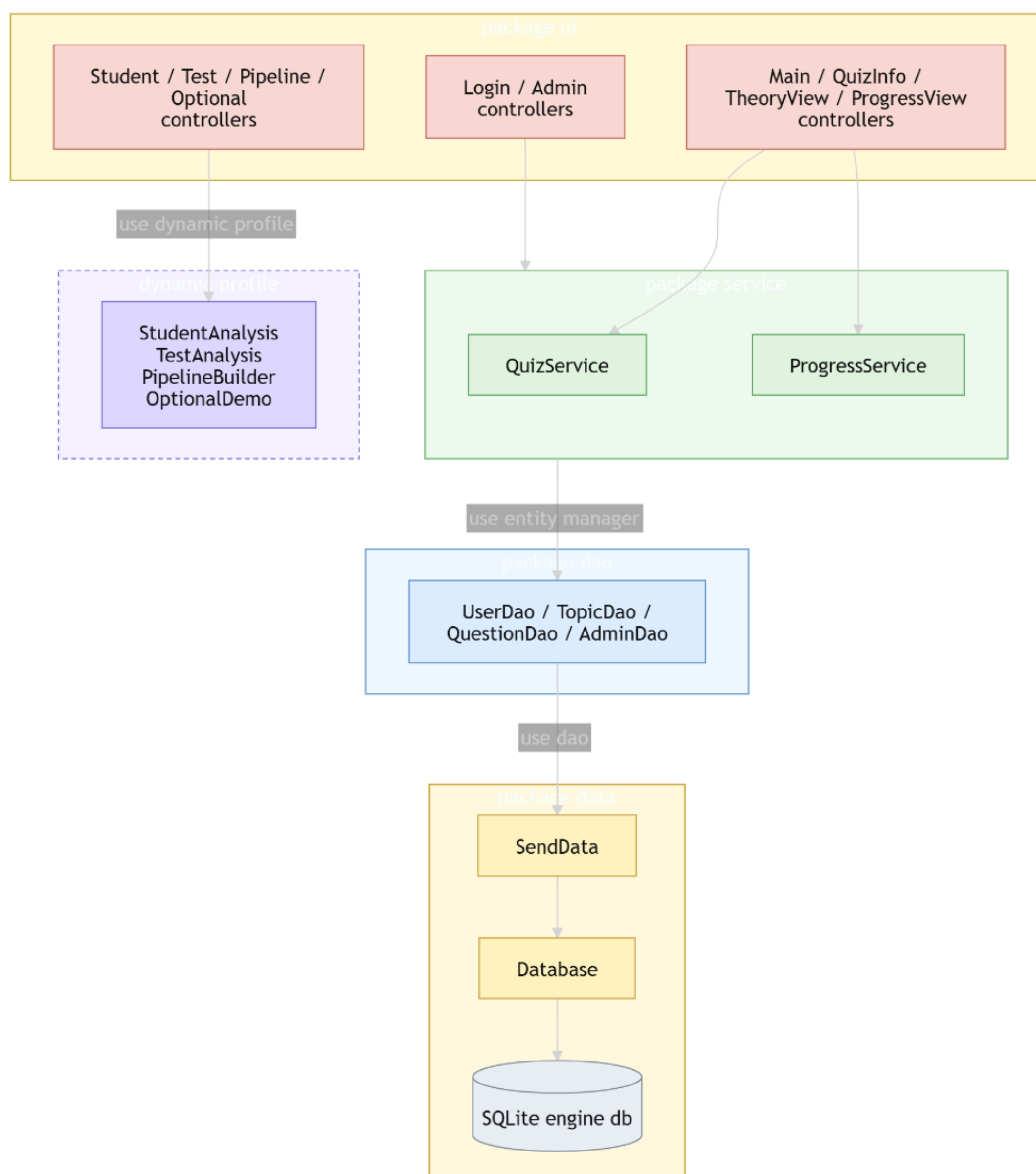

Рисунок 3.2 – Діаграма пакетів проєкту

Перелік класів за пакетами зведено у таблиці 3.2.

Таблиця 3.2 – Розподіл класів проєкту за пакетами

Пакет	**Кількість класів**	**Ключові класи**
model	8	User, Topic, Question, AnswerOption, Attempt, AttemptAnswer, Student, Step
db	2	Database, SeedData
dao	4	UserDao, TopicDao, QuestionDao, AttemptDao

Продовження таблиці 3.2

service	3	QuizService, ProgressService, ExportService
functional	4	StudentsAnalytics, TextAnalytics, PipelineBuilder, OptionalDemo
ui	10	LoginController, MainController, StudentsTabController, TextTabController, PipelineTabController, OptionalTabController, QuizTabController, ProgressTabController, TheoryTabController, AdminTabController
кореневий	2	App, Launcher

Проєкт містить 33 Java-файли, 10 FXML-розміток, 1 CSS-файл стилів. Така модульність забезпечує виконання нефункціональної вимоги НФ6 (розширюваність): додавання нової теми потребує лише розширення SeedData; додавання нового типу питання – розширення переліку Question.Type та відповідної гілки у QuizTabController.renderQuestion і QuizService.checkAnswer.

3.2. Алгоритмізація роботи застосунку

Тренажер реалізує чотири навчальні сценарії (демонстрації концепцій ФП) та один контрольний сценарій (тренування з банком питань). У цьому підрозділі формалізовано алгоритми ключових сценаріїв у вигляді блок-схем та описів кроків.

Сценарій 1. Демонстрація Stream API. Користувач працює з табличною колекцією студентів (CRUD-операції: додавання, видалення, скидання до зразкового набору). Обирає одну з семи Stream-операцій (filter+sorted, topByAverage, groupByCity, groupByGroup, averageByCity, countByGroup, overallSummary). Система виконує відповідний ланцюг викликів Stream API, відображає результат у текстовій панелі та показує еквівалентний фрагмент коду. Алгоритм цього сценарію наведено на рисунку2.3.

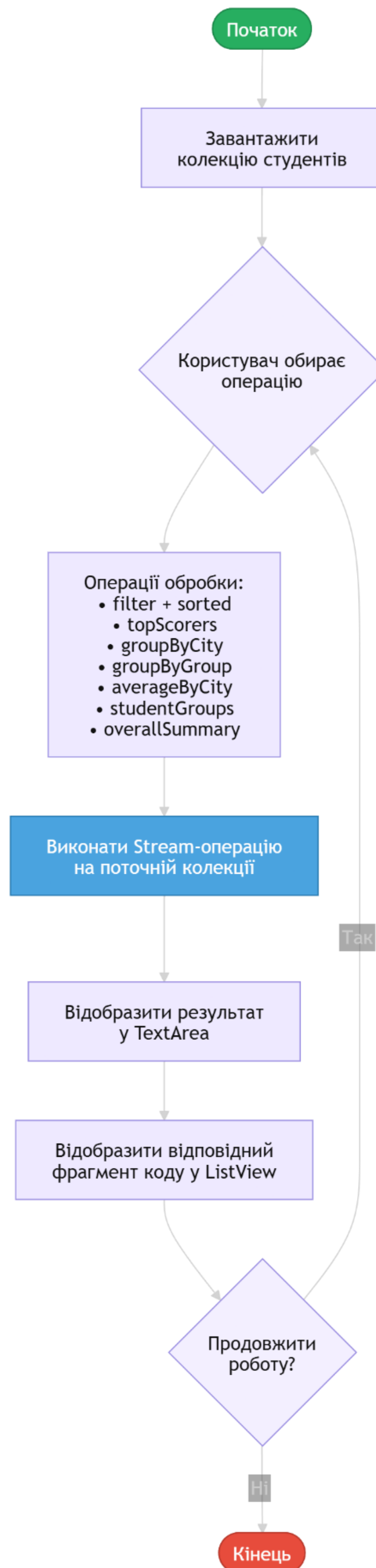

Рисунок 3.3 – Блок-схема сценарію «Демонстрація Stream API»

Усі сім Stream-операцій реалізовано як статичні чисті методи класу StudentsAnalytics. Кожен метод приймає список студентів і повертає нову

колекцію або значення, не змінюючи оригінал. Нижче у таблиці 3.3 наведено перелік методів з описом застосованих засобів ФП.

Таблиця 3.3 – Методи модуля StudentsAnalytics та використані засоби ФП

Метод	Опис	Засоби ФП
filterAndSortByAverage	Студенти із середнім балом вище порога, за спаданням	filter, sorted (Comparator), collect
topByAverage	Топ-N студентів за середнім балом	sorted, limit, collect
groupByCity	Групування студентів за містом	Collectors.groupingBy, LinkedHashMap
groupByGroup	Групування за навчальною групою	Collectors.groupingBy
averageByCity	Середній бал у розрізі міст	Collectors.groupingBy, averagingDouble
countByGroup	Кількість студентів за групами	Collectors.groupingBy, counting
overallSummary	Агрегована статистика по всіх оцінках	flatMapToInt, summaryStatistics

Сценарій 2. Аналіз тексту. Користувач вводить довільний текст (або завантажує зразковий). Система токенізує текст через regex-розбиття із приведенням до нижнього регістру (Locale.of("uk", "UA")), обчислює частотний словник (Collectors.groupingBy + counting), відображає топ-N слів у TableView та BarChart, виводить чотири агрегованих показники: загальна кількість слів, унікальні слова, середня довжина слова, найдовше слово. Усі обчислення зосереджені у класі TextAnalytics і є чистими функціями [15].

Сценарій 3. Конструктор конвеєра. Користувач формує послідовність кроків із набору операцій (Step.Kind): множення на k, додавання k, віднімання k, модуль, заперечення, піднесення у квадрат, фільтр за порогом, фільтр парних, фільтр непарних. Кожен крок перетворюється на IntUnaryOperator через метод Step.asOperator(). Клас PipelineBuilder збирає кроки у складене перетворення через

послідовний виклик IntUnaryOperator.andThen (композиція), починаючи від IntUnaryOperator.identity() – нейтрального елемента. Алгоритм:

```
IntUnaryOperator fn = IntUnaryOperator.identity();
for (Step s : steps) {
    fn = fn.andThen(s.asOperator());
}
```

Результуюча функція fn застосовується до кожного елемента вхідного списку через stream().mapToInt(fn).boxed().collect(toList()). Окремо обчислюється статистика (count, min, max, avg, sum, product). Користувач може змінювати порядок кроків (moveUp / moveDown), видаляти окремі кроки та очищувати конвеєр. Текстовий опис конвеєра оновлюється динамічно через слухач на ObservableList<Step>.

Сценарій 4. Демонстрація Optional. Реалізовано чотири незалежні модулі: (1) parseInt – обертає Integer.parseInt у Optional, повертаючи Optional.empty() замість NumberFormatException; (2) safeDivide – ділення двох чисел через flatMap-ланцюг: a.flatMap(x -> b.flatMap(y -> y == 0 ? empty() : of(x / y))); (3) safeLength / safeUpperCase – перетворення рядка через Optional.ofNullable(s).map(...).filter(...); (4) orElseGet з Supplier – демонстрація лінивої підстановки. Усі методи зосереджені у класі OptionalDemo, є чистими і покриті тестами [16].

Сценарій 5. Контроль знань (тренажер). Це ключовий сценарій, для якого наведемо детальну блок-схему (Рисунок 2.4).

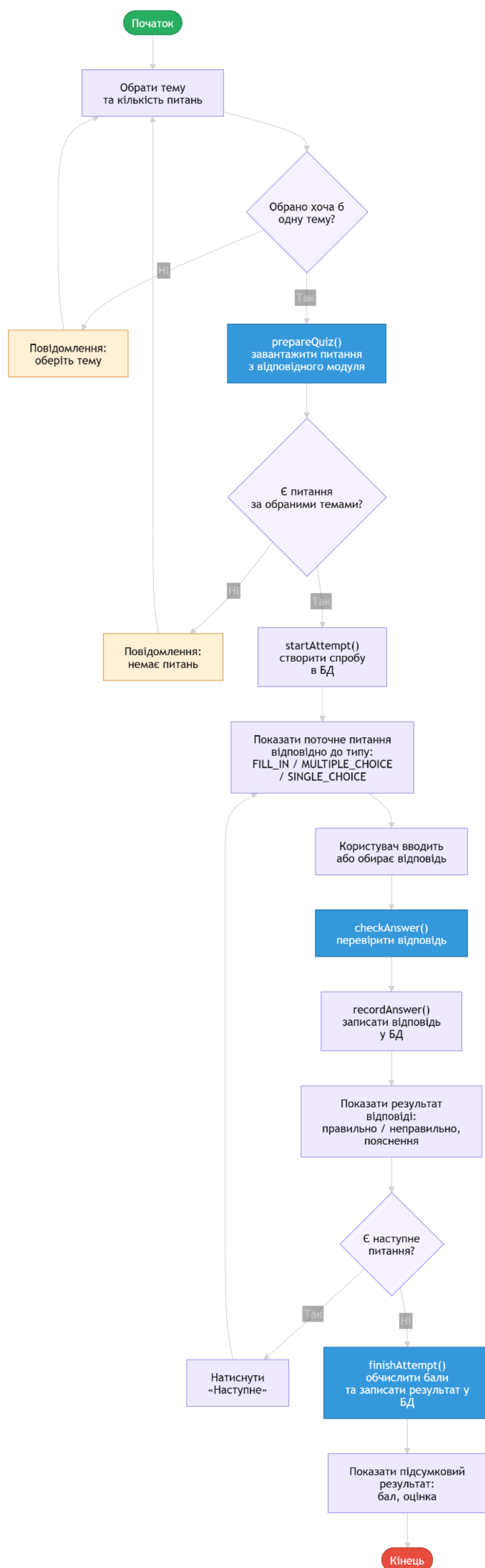

Рисунок 3.4 – Блок-схема сценарію «Проходження тренування»

Алгоритм сценарію «Проходження тренування» складається з таких кроків.

Крок 1. Формування набору питань. Користувач обирає одну або кілька тем (CheckBox) та кількість питань (Spinner). Метод QuizService.prepareQuiz завантажує питання з БД через QuestionDao.findByTopicIds, перемішує їх (Collections.shuffle), обмежує до заданої кількості. Для кожного питання також перемішуються варіанти відповідей – це унеможливлює запам'ятовування за позицією.

Крок 2. Створення запису спроби. Метод QuizService.startAttempt створює запис у таблиці attempts з поточним часом і ідентифікатором користувача.

Крок 3. Візуалізація питання. Контролер QuizTabController відображає поточне питання залежно від типу: для SINGLE_CHOICE – RadioButton з ToggleGroup, для MULTIPLE_CHOICE – CheckBox з підказкою «оберіть усі правильні», для FILL_IN – TextField з підказкою «регістр і пробіли не враховуються».

Крок 4. Перевірка відповіді. Метод QuizService.checkAnswer делегує перевірку одному з трьох приватних методів залежно від типу питання. Для SINGLE_CHOICE – порівнюється id обраного варіанта з id правильного. Для MULTIPLE_CHOICE – множина обраних id повинна точно збігатися з множиною правильних (часткова відповідь вважається невірною). Для FILL_IN – введений текст нормалізується (видаляються пробіли, крапки з комою, переводиться у нижній регістр) і порівнюється з еталоном [16].

Крок 5. Зворотний зв'язок. Після перевірки система показує: для правильної відповіді – зелений маркер та пояснення; для невірної – червоний маркер, текст правильної відповіді та пояснення. Пояснення зберігається у полі question.explanation і є частиною навчального ефекту – студент не лише дізнається, що помилився, а й чому.

Крок 6. Завершення спроби. Після останнього питання метод QuizService.finishAttempt обчислює бал як відношення кількості правильних відповідей до загальної кількості, помножене на 100, округлене до одного десяткового знаку. Запис у таблиці attempts оновлюється: проставляється час

завершення, кількість питань, кількість правильних, бал. Показується підсумковий екран із текстовою оцінкою: «Чудово!» (≥90%), «Добре» (≥75%), «Задовільно» (≥60%), «Варто повторити матеріал» (<60%). Прогрес оновлюється автоматично на вкладці «Прогрес».

Окремо слід зафіксувати алгоритм нормалізації текстової відповіді (FILL_IN), оскільки він забезпечує справедливу перевірку незалежно від стилю введення:

```
private static String normalize(String s) {
    return s.replaceAll("\\s+", "")
            .replace(";", "")
            .toLowerCase().trim();
}
```

Такий підхід дозволяє прийняти відповідь «(a,b) -> a + b» як еквівалентну «(a,b)->a+b» – студент не штрафується за форматування, а оцінюється лише за суть відповіді.

Реляційна модель даних тренажера складається з шести таблиць, які забезпечують зберігання профілів користувачів, навчальних тем, банку питань з варіантами відповідей, журналу спроб та окремих відповідей. Зв'язки між таблицями реалізовано через зовнішні ключі (FOREIGN KEY) із каскадним видаленням (ON DELETE CASCADE), що гарантує цілісність даних при видаленні батьківського запису.

Діаграму «сутність – зв'язок» (ER-діаграму) бази даних наведено на рисунку 3.5.

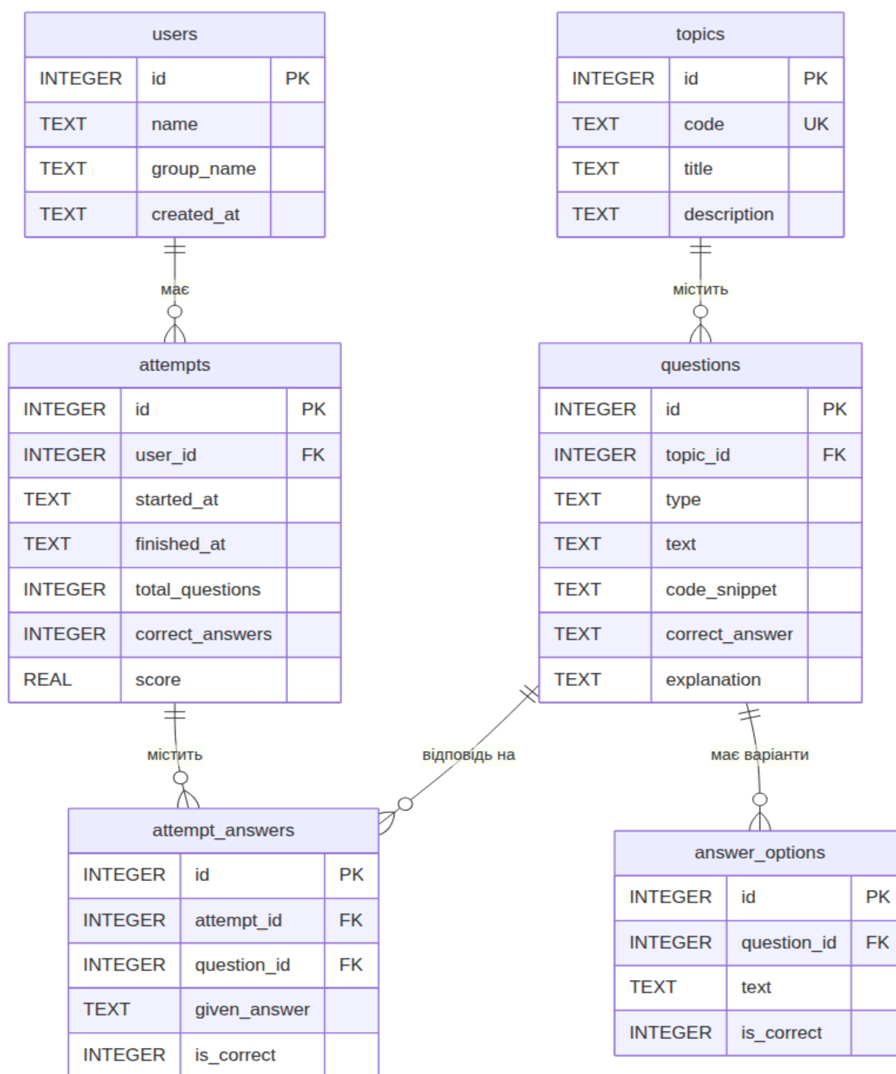

Рисунок 3.5 – ER-діаграма бази даних тренажера

Розглянемо кожну таблицю детально.

Таблиця users зберігає профілі студентів. Поле name (TEXT NOT NULL) містить прізвище та ім'я; group_name (TEXT, може бути NULL) – навчальну групу; created_at (TEXT NOT NULL) – дату і час створення у форматі ISO 8601 (LocalDateTime.toString()). Первинний ключ – id (INTEGER, AUTOINCREMENT).

Таблиця topics містить перелік навчальних тем. Поле code (TEXT NOT NULL UNIQUE) – унікальний ідентифікатор теми (LAMBDA, STREAM, OPTIONAL, COMPOSITION, COLLECTORS); title – назва для відображення; description – короткий опис. Унікальність code забезпечується обмеженням UNIQUE.

Таблиця questions – банк питань. Кожне питання прив'язане до теми через зовнішній ключ topic_id. Поле type (TEXT NOT NULL) визначає тип питання: SINGLE_CHOICE, MULTIPLE_CHOICE або FILL_IN. Поле text – текст питання; code_snippet – необов'язковий фрагмент коду, що відображається під текстом; correct_answer – еталонна відповідь для типу FILL_IN (для інших типів NULL); explanation – текст пояснення, який показується після перевірки.

Таблиця answer_options зберігає варіанти відповідей для питань типу SINGLE_CHOICE та MULTIPLE_CHOICE. Зв'язок із questions – через question_id. Поле text – текст варіанту; is_correct (INTEGER, 0 або 1) – ознака правильності. Для SINGLE_CHOICE рівно один варіант має is_correct = 1; для MULTIPLE_CHOICE – один або більше.

Таблиця attempts фіксує спроби проходження тренажера. Зв'язок із users – через user_id. Поля started_at і finished_at (TEXT) – час початку та завершення; total_questions і correct_answers (INTEGER) – кількість питань і правильних відповідей; score (REAL) – бал у відсотках. При створенні спроби поля finished_at, total_questions, correct_answers, score мають початкові значення (NULL або 0), які оновлюються після завершення [17].

Таблиця attempt_answers зберігає окремі відповіді користувача в межах конкретної спроби. Має два зовнішні ключі: attempt_id та question_id. Поле given_answer – введена/обрана відповідь у текстовому вигляді; is_correct – результат перевірки.

Опис полів усіх таблиць зведено у таблиці 3.4.

Таблиця 3.4 – Структура таблиць бази даних

Таблиця	Поле	Тип	Обмеження	Опис
users	id	INTEGER	PK, AI	Ідентифікатор
	name	TEXT	NOT NULL	Прізвище та ім'я
	group_name	TEXT		Навчальна група
	created_at	TEXT	NOT NULL	Дата створення (ISO 8601)

Продовження таблиці 3.4

topics	id	INTEGER	PK, AI	Ідентифікатор
	code	TEXT	NOT NULL, UK	Код теми (LAMBDA, STREAM тощо)
	title	TEXT	NOT NULL	Назва теми
	description	TEXT		Опис теми
questions	id	INTEGER	PK, AI	Ідентифікатор
	topic_id	INTEGER	FK → topics	Тема
	type	TEXT	NOT NULL	Тип: SINGLE / MULTIPLE / FILL_IN
	text	TEXT	NOT NULL	Текст питання
	code_snippet	TEXT		Фрагмент коду
	correct_answer	TEXT		Еталон для FILL_IN
	explanation	TEXT		Пояснення
answer_options	id	INTEGER	PK, AI	Ідентифікатор
	question_id	INTEGER	FK → questions	Питання
	text	TEXT	NOT NULL	Текст варіанта
	is_correct	INTEGER	NOT NULL	Правильність (0/1)
attempts	id	INTEGER	PK, AI	Ідентифікатор
	user_id	INTEGER	FK → users	Користувач
	started_at	TEXT	NOT NULL	Час початку
	finished_at	TEXT		Час завершення
	total_questions	INTEGER	NOT NULL	Кількість питань
	correct_answers	INTEGER	NOT NULL	Правильних відповідей
	score	REAL	NOT NULL	Бал (%)
attempt_answers	id	INTEGER	PK, AI	Ідентифікатор
	attempt_id	INTEGER	FK → attempts	Спроба

Продовження таблиці 3.4

	question_id	INTEGER	FK → questions	Питання
	given_answer	TEXT		Дана відповідь
	is_correct	INTEGER	NOT NULL	Результат (0/1)

Для прискорення запитів створено три індекси:

– idx_questions_topic – за полем topic_id таблиці questions (прискорює вибірку питань за обраними темами);
– idx_attempts_user – за полем user_id таблиці attempts (прискорює завантаження спроб конкретного користувача);
– idx_attempt_answers_attempt – за полем attempt_id таблиці attempt_answers (прискорює завантаження окремих відповідей спроби).

Підтримка зовнішніх ключів у SQLite [18] вимкнена за замовчуванням. Для її увімкнення при кожному відкритті з'єднання виконується PRAGMA foreign_keys = ON. Це забезпечує каскадне видалення: при видаленні теми автоматично видаляються всі її питання та варіанти; при видаленні користувача – всі його спроби та відповіді.

Заливання початкових даних реалізовано у класі SeedData. При першому запуску (коли таблиці topics і questions порожні) автоматично створюються п'ять тем та 32 питання (7 – LAMBDA, 7 – STREAM, 6 – OPTIONAL, 5 – COMPOSITION, 7 – COLLECTORS) з варіантами відповідей. Розподіл типів питань у банку: 20 SINGLE_CHOICE, 5 MULTIPLE_CHOICE, 7 FILL_IN. Такий баланс забезпечує різноманіття форм контролю.

DDL-скрипт (Data Definition Language) бази даних подано у Додатку Б бакалаврської роботи.

3.3. Проєктування застосунку

Графічний інтерфейс тренажера побудований за шаблоном MVC, у якому декларативна розмітка (FXML) відокремлена від логіки обробки подій (Java-контролери) і стилів (CSS) [19]. Інтерфейс складається з двох екранів: екрану входу та головного вікна з вісьмома вкладками. Схему навігації між екранами та вкладками наведено на рисунку 3.6.

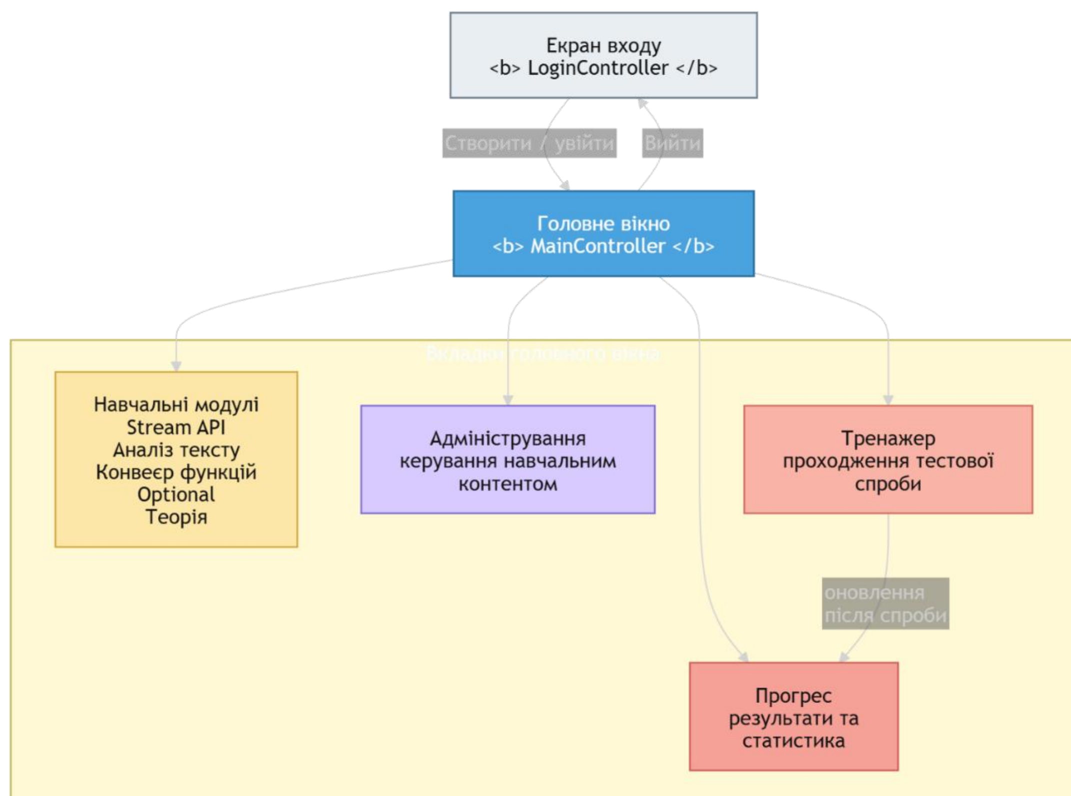

Рисунок 3.6 – Схема навігації інтерфейсу тренажера

Екран входу (Login.fxml + LoginController). Перший екран, який бачить користувач при запуску. Містить два сценарії входу: (1) створення нового профілю – поля «Прізвище та ім'я» і «Група» з кнопкою «Створити та увійти»; (2) вибір існуючого профілю зі списку ChoiceBox з кнопкою «Увійти». Профілі зберігаються в таблиці users. Після входу відкривається головне вікно; кнопка «Вийти» повертає на екран входу.

Головне вікно (Main.fxml + MainController). Містить верхню панель з назвою програми, ім'ям користувача та кнопкою «Вийти», а також TabPane з

вісьмома вкладками. Вкладки не можна закривати (tabClosingPolicy = UNAVAILABLE). Контент кожної вкладки завантажується з окремого FXML-файлу. Поділ вкладок на функціональні групи:

– Демонстраційні: «Stream API», «Аналіз тексту», «Конвеєр функцій», «Optional» – інтерактивні модулі, де студент експериментує з концепціями ФП.
– Контрольні: «Тренажер», «Прогрес» – контроль знань та відстеження результатів.
– Допоміжні: «Теорія», «Адмінування» – довідник та керування банком питань.

Розглянемо проєктні рішення для кожної вкладки.

Вкладка «Stream API» (StudentsTab.fxml). Побудована як SplitPane з двома панелями. Ліва панель містить таблицю студентів (TableView з колонками ПІБ, Місто, Група, Оцінки, Середній бал) та форму додавання (чотири текстових поля з кнопками «Додати», «Видалити обраного», «Скинути до прикладу»). Права панель містить елементи керування операціями: Spinner для порога та топ-N, сім кнопок з назвами відповідних операцій Stream API (filter+sorted, Топ-N, groupingBy(city), groupingBy(group), averagingDouble, counting, summaryStatistics), TextArea для результату і ListView для фрагмента коду.

Вкладка «Аналіз тексту» (TextTab.fxml). SplitPane: ліва частина – TextArea для введення тексту, кнопки «Аналізувати», «Завантажити приклад», «Очистити», Spinner для кількості слів у топі, GridPane з чотирма показниками. Права частина – TableView частотного словника (слово – частота), BarChart із гістограмою топ-N найчастіших слів [20].

Вкладка «Конвеєр функцій» (PipelineTab.fxml). Вертикальна компоновка VBox. Поле введення вхідних чисел; ChoiceBox для вибору типу кроку з 9 операцій Step.Kind; Spinner для параметра k; кнопка «Додати крок». ListView відображає поточну послідовність кроків із кнопками пересування (вгору, вниз) та видалення. Текстовий підпис динамічно оновлює формулу конвеєра (наприклад,

«x → (x → x * 2) → (x → x + 1)»). Кнопка «Застосувати конвеєр» виконує обчислення і відображає результат у TextArea.

Вкладка «Optional» (OptionalTab.fxml). Чотири «картки» (VBox зі стилем card), кожна присвячена окремому патерну: parseInt, safeDivide, safeLength/safeUpperCase, orElseGet. Кожна картка має текстові поля введення, кнопку виклику, мітку результату. Внизу – TextArea-журнал, де фіксуються всі виклики з параметрами та результатами.

Вкладка «Тренажер» (QuizTab.fxml). Побудована як ScrollPane з трьома станами видимості. Стан 1 (вибір параметрів): VBox із CheckBox для кожної теми, Spinner кількості питань, кнопка «Розпочати тренування». Стан 2 (питання): мітка прогресу, текст питання, мітка коду, VBox для варіантів (RadioButton / CheckBox / TextField залежно від типу), мітка зворотного зв'язку (зелена/червона), кнопки «Перевірити» і «Наступне». Стан 3 (результат): бал, текстова оцінка, кнопка «Розпочати ще раз». Перемикання між станами реалізовано через setVisible / setManaged.

Вкладка «Прогрес» (ProgressTab.fxml). GridPane із чотирма зведеними показниками (усього спроб, середній бал, найкращий, останній). TableView з журналом спроб (дата початку, дата завершення, кількість питань, правильних, бал). LineChart – лінійний графік динаміки балів за спробами. Кнопки «Оновити» та «Експорт у CSV» (FileChooser для збереження).

Вкладка «Теорія» (TheoryTab.fxml). SplitPane: зліва – ListView із переліком тем; справа – ScrollPane з TextFlow, у якому відображається форматований теоретичний матеріал (заголовки, абзаци, фрагменти коду шрифтом Consolas, маркери, примітки). Контент формується програмно у TheoryTabController без зовнішніх HTML-файлів.

Вкладка «Адмінування» (AdminTab.fxml). SplitPane: зліва – TableView з переліком питань (id, тема, тип, текст) та кнопки «Нове питання», «Видалити». Справа – форма редагування: ChoiceBox теми та типу, TextArea тексту питання та коду, TextField для еталонної відповіді (FILL_IN), TextArea пояснення. Нижче – TableView варіантів відповідей (editable: текст редагується через

TextFieldTableCell, ознака правильності – через CheckBoxTableCell), кнопки додавання та видалення варіантів, кнопка «Зберегти» з валідацією.

У таблиці 3.5 зведено відповідність між FXML-розмітками, контролерами та ключовими елементами інтерфейсу.

Таблиця 3.5 – Відповідність FXML-файлів, контролерів та ключових елементів GUI

FXML-файл	Контролер	Ключові елементи
Login.fxml	LoginController	TextField (ім'я, група), ChoiceBox, 2 кнопки
Main.fxml	MainController	TabPane (8 вкладок), Label, кнопка «Вийти»
StudentsTab.fxml	StudentsTabController	TableView, 4 TextField, 7 кнопок, TextArea, ListView
TextTab.fxml	TextTabController	TextArea, Spinner, TableView, BarChart, 4 Label
PipelineTab.fxml	PipelineTabController	TextField, ChoiceBox, Spinner, ListView, TextArea
OptionalTab.fxml	OptionalTabController	6 TextField, 5 Button, 5 Label, TextArea
QuizTab.fxml	QuizTabController	VBox (динамічні CheckBox/RadioButton/TextField), 3 стани
ProgressTab.fxml	ProgressTabController	GridPane, TableView, LineChart, FileChooser
TheoryTab.fxml	TheoryTabController	ListView, ScrollPane, TextFlow
AdminTab.fxml	AdminTabController	2 TableView, 2 ChoiceBox, 3 TextArea, editable cells

Стилізація. Усі елементи інтерфейсу стилізовані через файл app.css. Палітра витримана в академічному стилі: основний фон #f5f7fa (світло-сірий), верхня панель #243b53 (темний синій), акцентні кнопки #486581 (сіро-синій), зелений маркер правильної відповіді #2e7d32, червоний маркер помилки #c62828. Стиль-класи: .primary (акцентна кнопка), .success (правильно), .error (невірно), .mono (моноширинний текст), .card (картка з рамкою і закругленням), .question (текст питання 14 pt bold). Оформлення таблиць: заголовки з фоном #d9e2ec, виділення рядків при виборі #c6dafc [20].

РОЗДІЛ 4. ПРАКТИЧНА ЧАСТИНА

4.1. Реалізація функціоналу застосунку

У цьому підрозділі розглянуто програмну реалізацію чотирьох демонстраційних модулів тренажера та екрану входу. Для кожного модуля наведено ключові фрагменти коду з поясненнями, описано принцип взаємодії контролера з функційним шаром та подано екранні форми працюючого застосунку.

Екран входу. При запуску тренажера користувач бачить екран входу (Рисунок 4.1), який реалізовано у парі Login.fxml + LoginController. Екран пропонує два сценарії: створити новий профіль (ввести прізвище та ім'я, опційно – групу) або обрати існуючий профіль зі списку ChoiceBox, який заповнюється з таблиці users через UserDao.findAll(). Після входу ідентифікатор користувача передається у MainController через колбек Consumer<User>, що є прикладом функційного стилю навіть у навігаційній логіці.

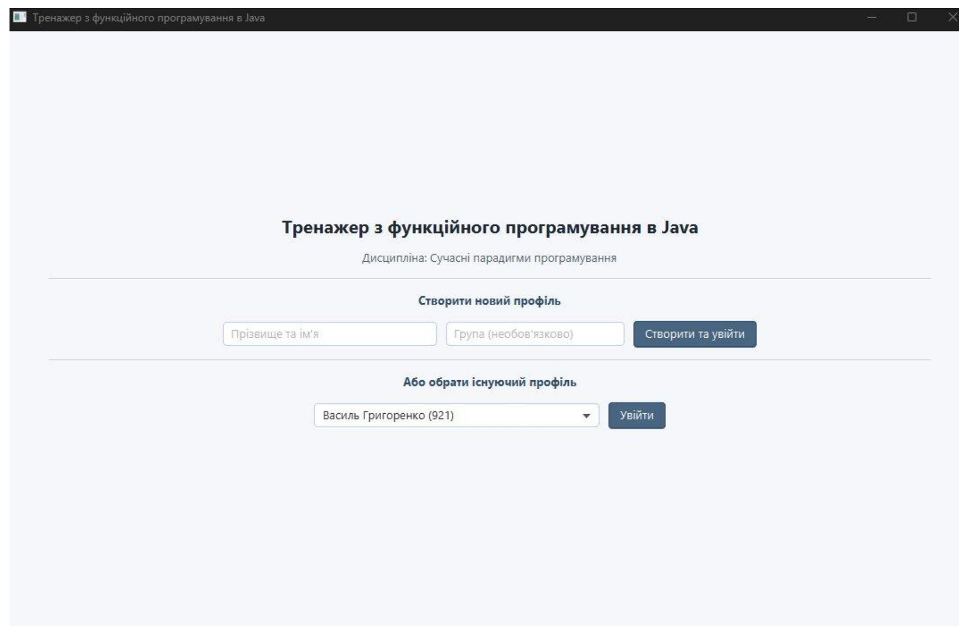

Рисунок 4.1 – Екран входу до тренажера

Метод setOnLoggedIn приймає Consumer<User> – функціональний інтерфейс, який MainController реалізує лямбда-виразом. Це дозволяє LoginController не мати жорсткої залежності від MainController:

```
// App.java
LoginController c = loader.getController();
c.setOnLoggedIn(this::showMain);  // method reference
```

Модуль «Stream API» (StudentsTabController + StudentsAnalytics). Цей модуль є центральним з навчальної точки зору, оскільки демонструє найширший спектр засобів Stream API. Інтерфейс побудований як SplitPane: ліва частина – таблиця студентів із формою редагування, права – панель операцій із результатом і фрагментом коду (Рисунок 4.2).

Рисунок 4.2 – Вкладка «Stream API» з таблицею студентів та операціями

Контролер StudentsTabController працює з ObservableList<Student>, яку ініціалізує зразковими даними через StudentsAnalytics.sampleStudents(). При натисканні кнопки відповідної операції контролер викликає статичний метод StudentsAnalytics, передаючи копію поточного списку. Результат відображається у

TextArea, а еквівалентний фрагмент коду – у ListView. Розглянемо реалізацію ключових операцій.

Операція filter + sorted реалізована методом filterAndSortByAverage, який демонструє ланцюжок filter → sorted → collect:

```
public static List<Student> filterAndSortByAverage(
        List<Student> students, double threshold) {
    return students.stream()
        .filter(s -> s.averageGrade() >= threshold)
        .sorted(Comparator.comparingDouble(
            Student::averageGrade).reversed())
        .collect(Collectors.toList());
}
```

Тут Predicate<Student> (лямбда s -> s.averageGrade() >= threshold) фільтрує студентів за порогом; Comparator.comparingDouble з method reference Student::averageGrade та reversed() задає зворотне сортування; collect(toList()) збирає результат у новий список. Оригінальний список students не модифікується – метод є чистою функцією [21].

Операція groupByCity демонструє Collectors.groupingBy з явним вказанням типу Map для збереження порядку ключів:

```
public static Map<String, List<Student>> groupByCity(
        List<Student> students) {
    return students.stream()
        .collect(Collectors.groupingBy(
            Student::getCity,
            LinkedHashMap::new,
            Collectors.toList()));
}
```

Трьохаргументна форма groupingBy приймає: (1) функцію-класифікатор Student::getCity, (2) supplier контейнера LinkedHashMap::new (для збереження порядку вставки), (3) вкладений колектор toList(). Цей приклад демонструє студентам композицію колекторів – один із найскладніших для розуміння аспектів Stream API.

Операція overallSummary демонструє flatMap та summaryStatistics – плоске розгортання вкладених списків оцінок усіх студентів із наступною агрегацією:

```
public static IntSummary overallSummary(List<Student> students) {
    var stats = students.stream()
        .flatMapToInt(s -> s.getGrades().stream()
            .mapToInt(Integer::intValue))
        .summaryStatistics();
    return new IntSummary(stats.getCount(), stats.getMin(),
        stats.getMax(), stats.getAverage(), stats.getSum());
}
```

flatMapToInt перетворює Stream<Student> на IntStream усіх оцінок, а summaryStatistics повертає IntSummaryStatistics з count, min, max, average, sum. Результат обгортається у record IntSummary для зручного відображення.

Модуль «Аналіз тексту» (TextTabController + TextAnalytics). Модуль дозволяє студентові ввести довільний текст і побачити результати його аналізу засобами Stream API: частотний словник у вигляді таблиці, гістограму топ-N слів, агреговані показники (Рисунок 4.3).

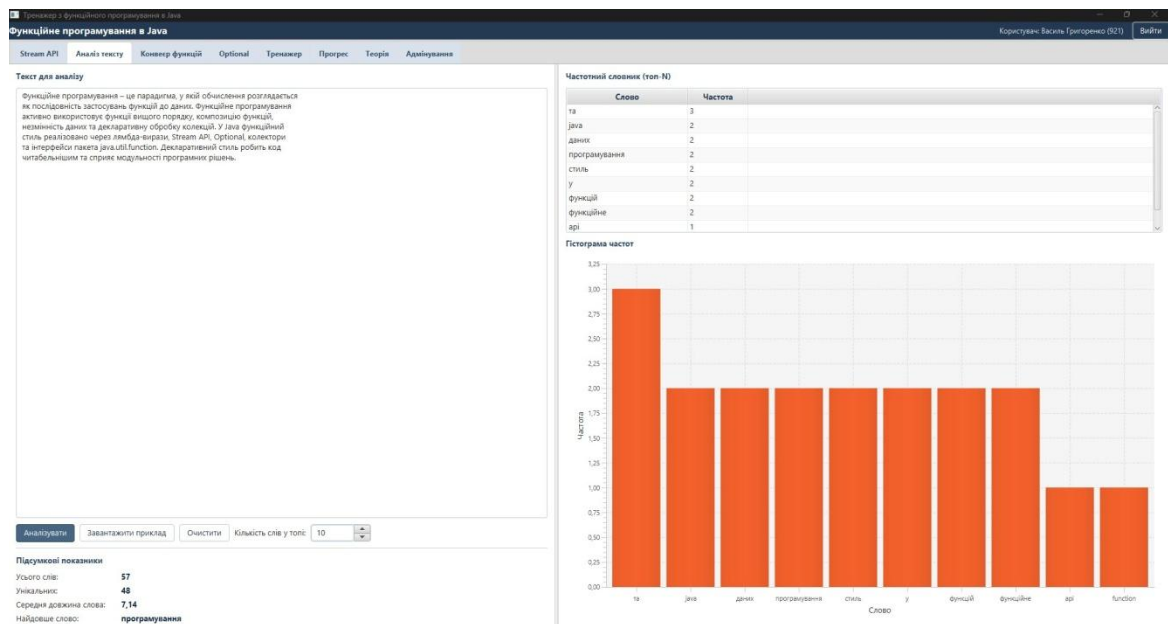

Рисунок 4.3 – Вкладка «Аналіз тексту» з частотним словником та гістограмою

Клас TextAnalytics містить шість чистих статичних методів. Базовим є tokenize – розбиття тексту на слова з приведенням до нижнього регістру:

```
public static List<String> tokenize(String text) {
    if (text == null || text.isBlank()) return List.of();
    return Arrays.stream(text.toLowerCase(
            Locale.of("uk", "UA")).split(WORD_DELIMITERS))
        .filter(s -> !s.isBlank())
        .collect(Collectors.toList());
}
```

Токенізація використовує Arrays.stream для створення стріму з масиву рядків після split, Predicate<String> для фільтрації порожніх токенів, Locale.of для коректного приведення регістру українських літер.

Частотний словник будується методом wordFrequencies, який демонструє groupingBy + counting + сортування за значенням:

```
public static Map<String, Long> wordFrequencies(String text) {
    return tokenize(text).stream()
        .collect(Collectors.groupingBy(w -> w, counting()))
        .entrySet().stream()
        .sorted(Map.Entry.<String, Long>comparingByValue()
            .reversed().thenComparing(
                Map.Entry.comparingByKey()))
        .collect(Collectors.toMap(
            Map.Entry::getKey, Map.Entry::getValue,
            (a, b) -> a, LinkedHashMap::new));
}
```

Цей метод є одним з найскладніших у проєкті: він демонструє двоетапний конвеєр (спочатку підрахунок частот через groupingBy + counting, потім сортування entrySet за значенням у зворотному порядку). Результат зберігається у LinkedHashMap для збереження порядку. Контролер TextTabController відображає результат у TableView та будує BarChart із серією XYChart.Series.

Модуль «Конвеєр функцій» (PipelineTabController + PipelineBuilder). Модуль демонструє фундаментальну ідею ФП – композицію функцій. Користувач формує послідовність кроків (операцій), які комбінуються через andThen у єдине перетворення. Інтерфейс (Рисунок 3.4) містить ChoiceBox для вибору з 9 типів

операцій (Step.Kind), Spinner для параметра k, ListView для візуалізації конвеєра та TextArea для результату.

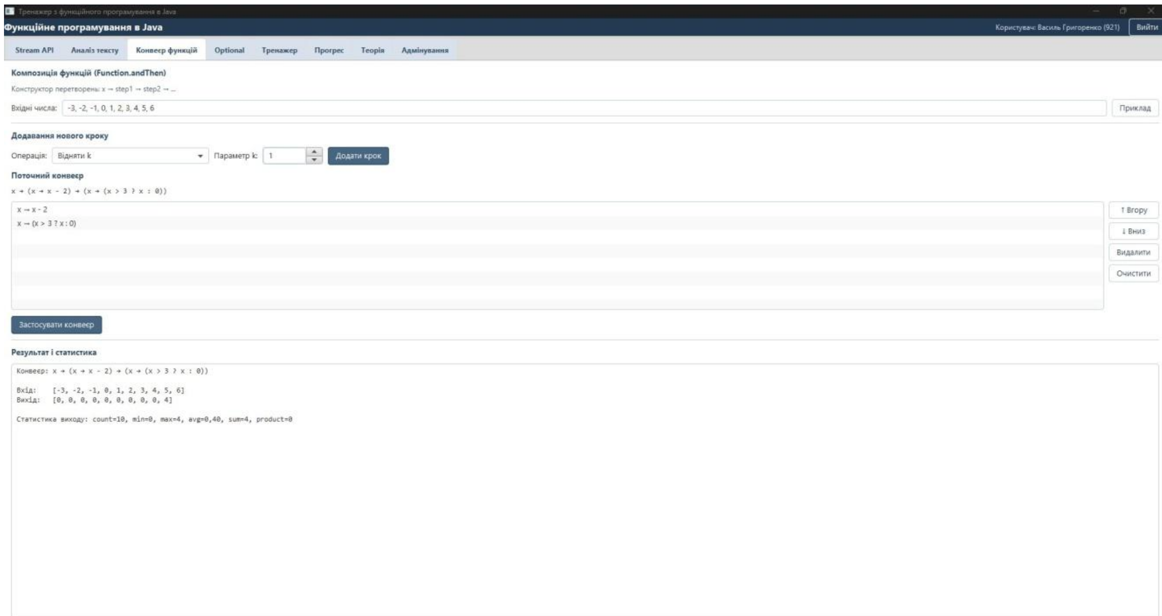

Рисунок 4.4 – Вкладка «Конвеєр функцій» із двома кроками та результатом обчислення

Кожний крок конвеєра представлено класом Step, який інкапсулює вид операції (Step.Kind) та параметр. Метод asOperator() повертає IntUnaryOperator через switch-вираз:

```
public IntUnaryOperator asOperator() {
    final int k = parameter;
    return switch (kind) {
        case MULTIPLY -> x -> x * k;
        case ADD      -> x -> x + k;
        case SQUARE   -> x -> x * x;
        case ABS      -> Math::abs;
        case NEGATE   -> x -> -x;
        case FILTER_GREATER -> x -> x > k ? x : 0;
        // ... інші операції
    };
}
```

Тут switch-вираз Java 17+ повертає лямбди та method references – кожна гілка генерує IntUnaryOperator. Зауважимо використання Math::abs як method reference на статичний метод [22].

Клас PipelineBuilder збирає кроки у складене перетворення через послідовне застосування andThen:

```
public static IntUnaryOperator combine(List<Step> steps) {
    IntUnaryOperator fn = IntUnaryOperator.identity();
    for (Step s : steps) {
        fn = fn.andThen(s.asOperator());
    }
    return fn;
}
```

Початковою точкою служить IntUnaryOperator.identity() – нейтральний елемент композиції (функція x → x). Кожне наступне andThen додає новий крок до ланцюга. Результуюча функція fn застосовується до кожного елемента вхідного списку через stream().mapToInt(fn).boxed().collect(toList()). На Рисунок 3.4 показано конвеєр із двох кроків: x → x − 2, потім x → (x > 3 ? x : 0). На вході [-3, -2, ..., 6], на виході більшість елементів перетворено на 0, і лише 4 (від вхідного 6: (6−2)=4, 4>3 → 4) залишається ненульовим.

Модуль «Optional» (OptionalTabController + OptionalDemo). Модуль складається з чотирьох незалежних секцій, кожна з яких ілюструє конкретний патерн роботи з Optional (Рисунок 4.5).

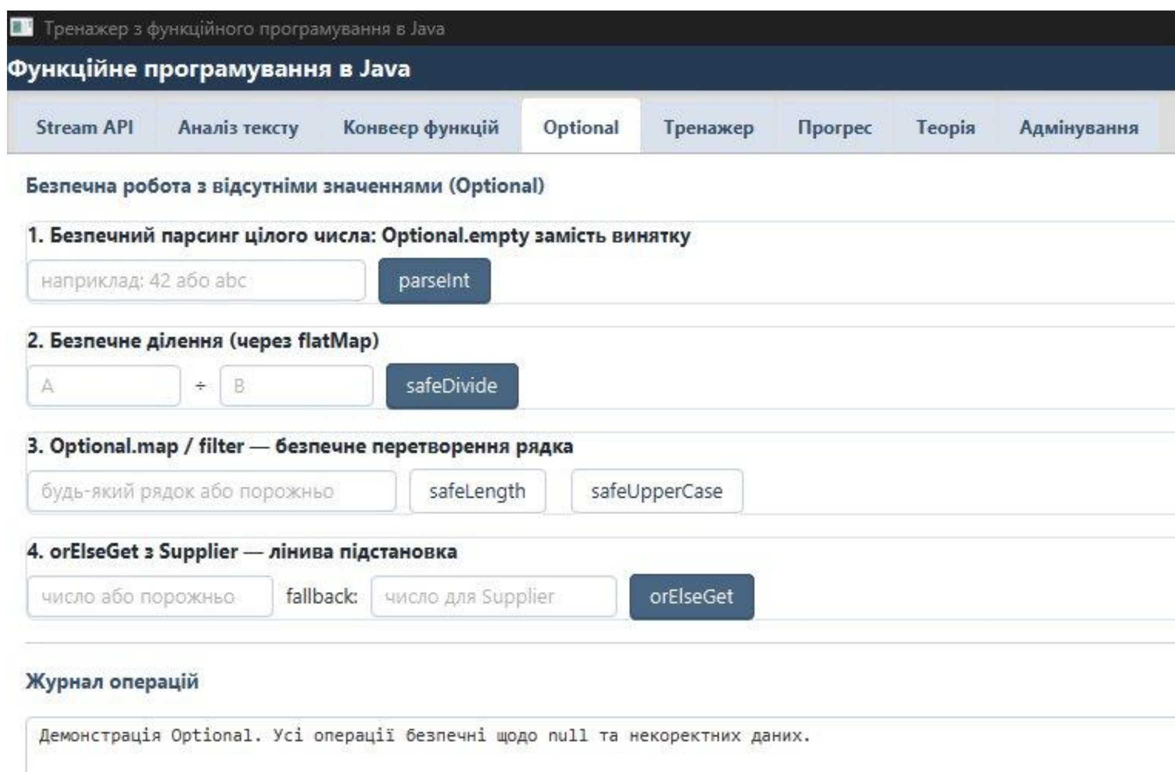

Рисунок 4.5 – Вкладка «Optional» із чотирма секціями демонстрацій

Секція 1 – parseInt: обертає Integer.parseInt у Optional, повертаючи Optional.empty() замість NumberFormatException. Секція 2 – safeDivide: реалізує ланцюг flatMap для безпечного ділення двох чисел з контролем нуля:

```
public static Optional<Double> safeDivide(String aText, String bText) {
    Optional<Double> a = parseDouble(aText);
    Optional<Double> b = parseDouble(bText);
    return a.flatMap(x -> b.flatMap(y ->
        y == 0 ? Optional.empty() : Optional.of(x / y)));
}
```

Тут flatMap вкладено двічі: зовнішній flatMap розгортає Optional<Double> a, внутрішній – b. Якщо будь-яке з них порожнє (некоректне введення) або b дорівнює нулю, результатом буде Optional.empty(). Студент бачить, як Optional елегантно замінює каскад if-null-перевірок.

Секція 3 демонструє Optional.map і filter: safeLength повертає довжину рядка через Optional.ofNullable(s).map(String::length).orElse(0); safeUpperCase

фільтрує порожні рядки через filter(str -> !str.isBlank()). Секція 4 – orElseGet із Supplier: демонструє лінивий підхід до підстановки альтернативного значення.

Усі методи OptionalDemo є чистими: вони не мають побічних ефектів, не модифікують стан, повертають Optional або примітив. Контролер OptionalTabController протоколює кожен виклик у журнал (TextArea logArea), що дозволяє студентові відстежити послідовність операцій.

Важливим аспектом реалізації всіх чотирьох модулів є відображення фрагментів коду. Коли студент натискає кнопку операції, контролер не лише показує результат, а й виводить відповідний фрагмент Java-коду [23] у ListView (для Stream API) або пояснювальний текст. Це створює зв'язок між візуальним результатом і кодом, який його породив, що є ключовим для навчального ефекту.

4.2. Реалізація підсистеми тренажера та графічного інтерфейсу

Підсистема тренажера є контрольною частиною програми: вона забезпечує перевірку засвоєння матеріалу через банк питань, фіксує результати у базі даних та надає засоби аналізу прогресу. У підрозділі розглянуто реалізацію QuizService, шар DAO [24], механізм ініціалізації БД, вкладки «Тренажер», «Прогрес» та «Адмінування».

Ініціалізація бази даних. Клас Database реалізує патерн Singleton (через synchronized-метод init()). При першому запуску створюється каталог ~/.fp-java-trainer/ та файл fpjava.db. Явне завантаження JDBC-драйвера через Class.forName("org.sqlite.JDBC") гарантує реєстрацію навіть при нестандартній конфігурації classpath. Після підключення виконується PRAGMA foreign_keys = ON та DDL-скрипт створення шести таблиць (якщо вони відсутні). Клас SeedData перевіряє, чи порожні таблиці topics і questions, і за потреби заливає 5 тем та 32 питання.

Шар DAO. Чотири класи DAO (UserDao, TopicDao, QuestionDao, AttemptDao) інкапсулюють усі SQL-запити. Кожен метод DAO відкриває та

закриває з'єднання у блоці try-with-resources. QuestionDao є найскладнішим: він підтримує повний CRUD (create, read, update, delete) для питань з каскадним збереженням варіантів відповідей. Метод findByTopicIds формує динамічний SQL із параметризованим IN-виразом:

```
StringBuilder sql = new StringBuilder("SELECT ... WHERE topic_id IN (");
for (int i = 0; i < topicIds.size(); i++)
    sql.append(i == 0 ? "?" : ",?");
sql.append(") ORDER BY id");
```

Це безпечний підхід – замість конкатенації значень у рядок використовуються іменовані параметри (?), що запобігає SQL-ін'єкціям.

QuizService – логіка тренажера. Клас QuizService координує весь процес тренування. Його основні методи:

- prepareQuiz(topicIds, limit) – завантажує питання з БД, перемішує їх (Collections.shuffle), обрізає до limit, перемішує варіанти відповідей для кожного питання.
- checkAnswer(question, givenAnswer) – делегує перевірку відповідному методу за типом (через switch-вираз).
- startAttempt(userId) – створює запис у таблиці attempts через AttemptDao.
- recordAnswer(attempt, question, answer, correct) – зберігає відповідь у attempt_answers.
- finishAttempt(attempt, total, correct) – обчислює бал та оновлює запис спроби.

Алгоритм перевірки для кожного типу питання:

Для SINGLE_CHOICE – введене значення (id обраного RadioButton) парситься як Long і порівнюється з id правильного варіанта в списку options. Використовується Optional-ланцюг:

```
return q.getOptions().stream()
    .filter(o -> o.getId() == chosenId)
    .findFirst()
    .map(AnswerOption::isCorrect)
    .orElse(false);
```

Для MULTIPLE_CHOICE – множина обраних id (Set<Long>) порівнюється з множиною правильних. Часткова відповідь вважається невірною:

```
Set<Long> chosen = parseIdSet(givenAnswer);
Set<Long> correct = q.getOptions().stream()
    .filter(AnswerOption::isCorrect)
    .map(AnswerOption::getId)
    .collect(Collectors.toCollection(HashSet::new));
return chosen.equals(correct);
```

Для FILL_IN – текст нормалізується (пробіли, крапки з комою видаляються, регістр ігнорується) і порівнюється з еталоном. Це дозволяє прийняти «(a, b) -> a + b» як еквівалент «(a,b)->a+b».

Вкладка «Тренажер» (QuizTabController). Контролер реалізує три стани: вибір параметрів (теми, кількість), процес відповідей, підсумковий екран (Рисунок 4.6). Перемикання стану здійснюється через setVisible/setManaged для відповідних VBox-панелей [25].

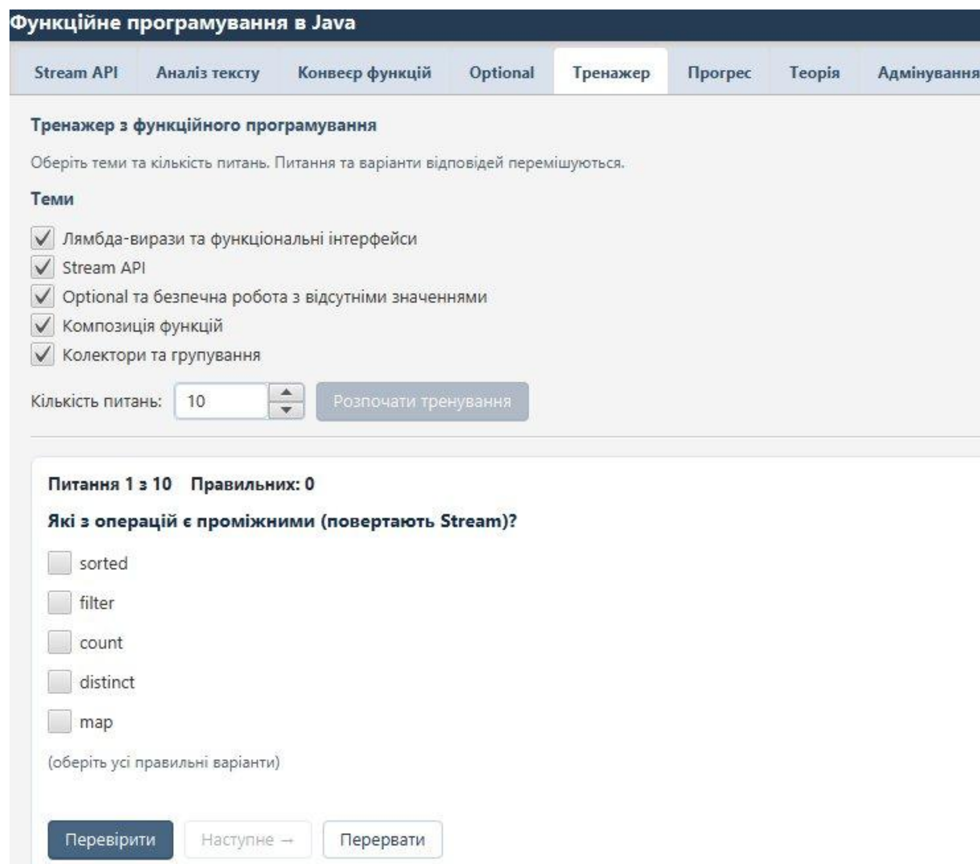

Рисунок 4.6 – Вкладка «Тренажер»: вибір тем та питання типу MULTIPLE_CHOICE

На рисунку 4.6 зображено питання типу MULTIPLE_CHOICE: система пропонує обрати операції, що повертають Stream (тобто є проміжними). Варіанти відображаються як CheckBox [26] (на відміну від RadioButton для SINGLE_CHOICE). Після натискання «Перевірити» контролер формує рядок з id обраних варіантів через:

```
Set<Long> ids = new TreeSet<>();
for (var n : optionsBox.getChildren()) {
    if (n instanceof CheckBox cb && cb.isSelected())
        ids.add((Long) cb.getUserData());
}
yield ids.stream().map(String::valueOf)
    .collect(Collectors.joining(","));
```

Зверніть увагу на використання pattern matching (n instanceof CheckBox cb) – засіб Java 17+, та Stream для формування CSV-рядка з Collectors.joining.

Після перевірки показується зворотний зв'язок: для правильної відповіді – зелений маркер (CSS-клас .success) з текстом пояснення; для невірної – червоний маркер (.error), текст правильної відповіді та пояснення. Стиль-класи додаються та видаляються динамічно через getStyleClass().removeAll("error") / getStyleClass().add("success").

Вкладка «Прогрес» (ProgressTabController). Вкладка (Рисунок 4.7) відображає зведену статистику за допомогою ProgressService та деталізований журнал спроб.

Рисунок 4.7 – Вкладка «Прогрес» зі зведеними показниками та журналом спроб

ProgressService обчислює статистику через Stream API:

```
public Stats stats(long userId) {
    List<Attempt> attempts = attempts(userId);
    long count = attempts.size();
    OptionalDouble avg = attempts.stream()
        .mapToDouble(Attempt::getScore).average();
    double best = attempts.stream()
        .mapToDouble(Attempt::getScore).max().orElse(0.0);
    double last = attempts.isEmpty() ? 0.0
        : attempts.get(0).getScore();
    return new Stats(count, avg.orElse(0.0), best, last);
}
```

Тут OptionalDouble – спеціалізація Optional для примітиву double. Метод average() повертає OptionalDouble (який може бути порожнім для порожнього стріму), а orElse(0.0) забезпечує безпечну підстановку. Графік динаміки будується через LineChart: для кожної спроби створюється XYChart.Data [27] з датою та балом.

Експорт у CSV реалізований у класі ExportService, який демонструє Collectors.joining:

```
String body = attempts.stream()
    .map(a -> Stream.of(
            String.valueOf(a.getId()),
            a.getStartedAt().format(FMT),
            String.valueOf(a.getScore()))
        .collect(Collectors.joining(";")))
    .collect(Collectors.joining("\n"));
```

Зовнішній стрім ітерує по спробах, внутрішній Stream.of формує рядок із полів через joining(";"), а зовнішній joining("\n") з'єднує рядки. Таким чином, навіть утилітарна операція експорту стає демонстрацією функційного стилю.

Вкладка «Адмінування» (AdminTabController). Вкладка (Рисунок 4.8) надає повний CRUD-інтерфейс для управління банком питань без перекомпіляції програми. Зліва – TableView з переліком усіх питань (32 за замовчуванням), справа – форма редагування.

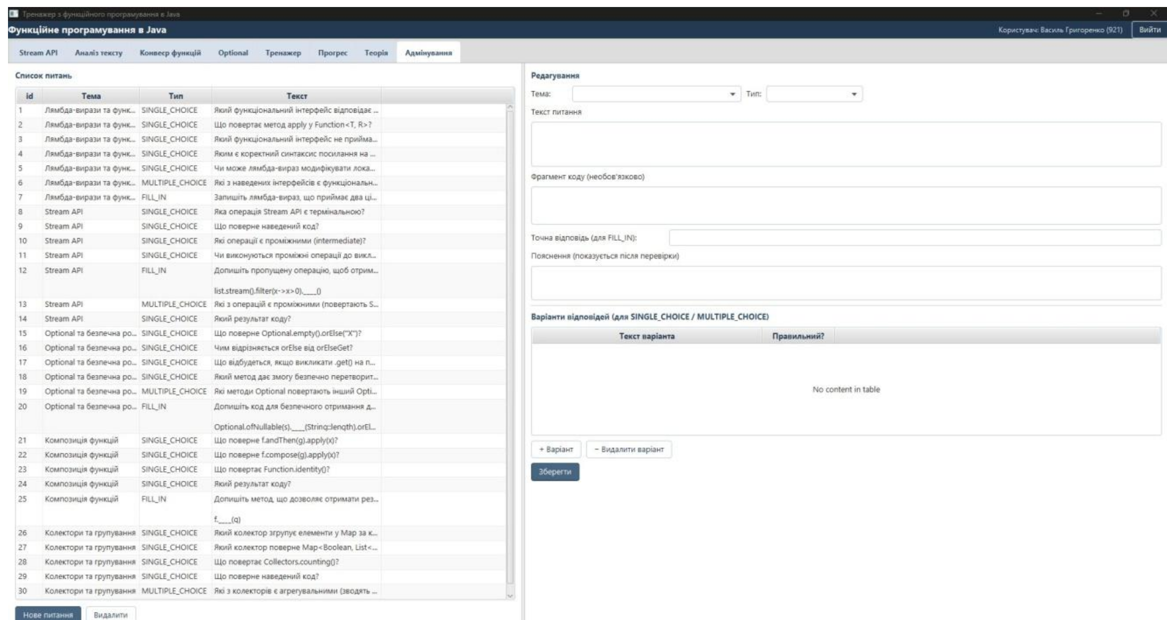

Рисунок 4.8 – Вкладка «Адмінування» з банком питань та формою редагування

Таблиця варіантів відповідей реалізована як editable TableView: текст варіанта редагується через TextFieldTableCell.forTableColumn, ознака

правильності – через CheckBoxTableCell.forTableColumn. При збереженні контролер виконує валідацію залежно від типу питання:

– для SINGLE_CHOICE – рівно один варіант має бути позначений як правильний;
– для MULTIPLE_CHOICE – хоча б один варіант правильний, мінімум два варіанти загалом;
– для FILL_IN – поле «Точна відповідь» не може бути порожнім.

QuestionDao.save використовує стратегію «insert or update»: якщо id = 0, створюється новий запис; якщо id > 0, оновлюється існуючий. При оновленні старі варіанти відповідей видаляються (DELETE FROM answer_options WHERE question_id = ?) і зберігаються заново. Це спрощує логіку порівняно з відстеженням змін у кожному окремому варіанті [28].

Вкладка «Теорія» (TheoryTabController). Реалізує вбудований довідник із п'яти тем. Контент формується програмно через масив Block-об'єктів, кожен з яких має тип (HEADING, PARAGRAPH, CODE, BULLET, NOTE) і текст. Метод showArticle рендерить блоки у TextFlow, динамічно створюючи об'єкти Text з різними шрифтами та стилями. Фрагменти коду виводяться шрифтом Consolas, заголовки – жирним Segoe UI, примітки – з символом □.

Графічний інтерфейс тренажера реалізовано за шаблоном MVC, у якому розмітка компонентів описана у FXML-файлах, стилі зосереджені у єдиному CSS-файлі app.css, а логіка обробки подій розміщена у Java-контролерах. Загалом проєкт містить 10 FXML-розміток (611 рядків), 1 CSS-файл (187 рядків) і 10 класів-контролерів. У підрозділі розглянуто ключові принципи реалізації GUI та наведено характерні фрагменти [29].

Завантаження інтерфейсу. Точкою входу графічної частини є клас App, який успадковує javafx.application.Application. У методі start виконується ініціалізація БД (Database.init()), завантаження FXML-розмітки екрану входу через FXMLLoader, підключення CSS-стилів та показ вікна. Перехід від екрану входу до головного вікна реалізовано через колбек Consumer<User>:

```
// App.java
LoginController c = loader.getController();
c.setOnLoggedIn(this::showMain);


// LoginController.java
private Consumer<User> onLoggedIn;
public void setOnLoggedIn(Consumer<User> cb) {
    this.onLoggedIn = cb;
}
```

Такий підхід демонструє функційний стиль навіть у навігаційній логіці: LoginController не має залежності від App – він лише приймає «що робити після входу» як значення-функцію.

Структура FXML. Кожна FXML-розмітка починається з директив імпортів JavaFX-класів, за якими слідує кореневий контейнер із вказанням контролера через атрибут fx:controller. Елементи інтерфейсу прив'язуються до полів контролера через fx:id, а обробники подій – через onAction="#methodName". Приклад структури Login.fxml:

```
<VBox fx:controller="...ui.LoginController"
      alignment="CENTER" spacing="14">
    <TextField fx:id="nameField" promptText="Прізвище та ім'я"/>
    <Button text="Створити та увійти"
            onAction="#onCreate" styleClass="primary"/>
    <ChoiceBox fx:id="usersChoice" prefWidth="320"/>
</VBox>
```

У контролері поля з анотацією @FXML автоматично ін'єктуються FXMLLoader під час завантаження:

```
@FXML private TextField nameField;
@FXML private ChoiceBox<User> usersChoice;


@FXML private void onCreate() {
    String name = nameField.getText().trim();
    // валідація, створення користувача...
}
```

Статична перевірка відповідності fx:id і @FXML-полів виконана спеціальним скриптом, який для кожного з 10 FXML-файлів [30] звіряє множину fx:id з множиною полів контролера та onAction з методами. Усі 10 файлів пройшли перевірку без розбіжностей.

Головне вікно і TabPane. Main.fxml побудований як BorderPane: верхня частина (top) – панель із заголовком програми, ім'ям користувача та кнопкою «Вийти»; центральна частина (center) – TabPane з восьмома вкладками. Вкладки не можна закривати (tabClosingPolicy="UNAVAILABLE"). Контент кожної вкладки завантажується в MainController.loadTabs() через FXMLLoader:

```
FXMLLoader quizLoader = new FXMLLoader(
    getClass().getResource("/...fxml/QuizTab.fxml"));
Node quizNode = quizLoader.load();
quizController = quizLoader.getController();
quizController.setUser(currentUser);
quizTab.setContent(quizNode);
```

Для вкладок «Тренажер» і «Прогрес» контролер зберігає посилання на об'єкти, щоб після завершення спроби автоматично оновити прогрес:

```
quizController.setOnAttemptFinished(
    () -> progressController.refresh());
```

Компоновка елементів. Для розміщення елементів використано стандартні контейнери JavaFX: SplitPane (вкладки Stream API, Аналіз тексту, Теорія, Адмінування – двопанельна компоновка), VBox (вертикальне розташування – Конвеєр, Optional), GridPane (зведені показники у Прогресі та Аналізі тексту), HBox (горизонтальне групування кнопок), ScrollPane (прокрутка у Тренажері і Теорії). Відступи задаються через тег <Insets> та атрибут spacing. Пропорції SplitPane контролюються атрибутом dividerPositions (наприклад, 0.55 для Stream API, 0.25 для Теорії) [31].

Вкладка «Теорія» – TextFlow замість WebView. У ранніх версіях теоретичний довідник реалізовувався через WebView (javafx.scene.web), однак цей модуль потребує jdk.jsobject, який видалено із JDK 24+. Для забезпечення сумісності з JDK 24–26 WebView замінено на TextFlow – легкий компонент для

форматованого тексту. Контент формується програмно через масив Block-об'єктів із типами HEADING, PARAGRAPH, CODE, BULLET, NOTE. Метод showArticle рендерить блоки у TextFlow, створюючи об'єкти Text із різними шрифтами (Segoe UI для тексту, Consolas для коду) та стилями. На Рисунок 3.9 показано результат рендеру статті «Optional» із заголовками, фрагментами коду та маркерами.

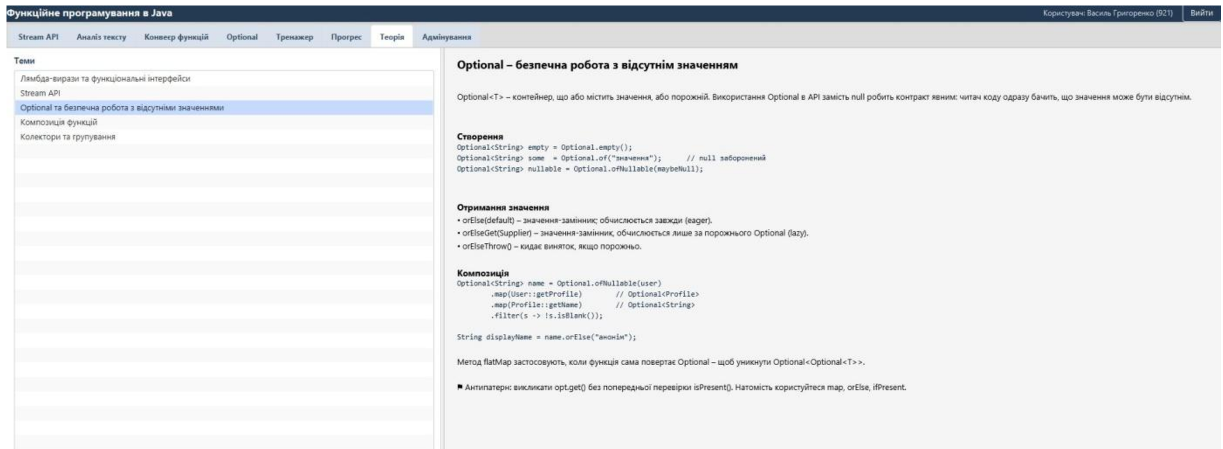

Рисунок 4.9 – Вкладка «Теорія» з теоретичним матеріалом по Optional

CSS-стилізація. Файл app.css (187 рядків) визначає палітру, стилі кнопок, таблиць, полів введення, маркерів правильних/неправильних відповідей. Палітра витримана в академічному стилі: основний фон #f5f7fa, верхня панель #243b53, акцентні кнопки #486581, успіх #2e7d32, помилка #c62828. Основні стиль-класи:

- .primary – акцентна кнопка (сіро-синій фон, білий текст, скруглені кути 4px);
- .success – зелений текст для правильної відповіді (використовується у feedbackLabel);
- .error – червоний текст для неправильної відповіді;
- .mono – моноширинний шрифт Consolas 12px для фрагментів коду;
- .card – блок із рамкою #d9e2ec та скругленням 6px (використовується у Optional і Тренажері);
- .question – текст питання 14pt bold для виділення у тренажері;
- .code-block – фон #f0f4f8 з padding 8px для блоків коду в тренажері.

CSS підключається до сцени у класі App:

```
scene.getStylesheets().add(Objects.requireNonNull(
    getClass().getResource("/...css/app.css"))
    .toExternalForm());
```

Динамічне перемикання стилів використовується у QuizTabController для зворотного зв'язку:

```
feedbackLabel.getStyleClass().removeAll("error");
feedbackLabel.getStyleClass().add("success");
```

Обробка таблиць. У проєкті використовується 6 TableView (студенти, частотний словник, журнал спроб, адмін-питання, адмін-варіанти, кроки конвеєра). Зв'язок TableView з моделлю реалізовано через setCellValueFactory – прив'язку колонки до властивості моделі. Для простих JavaBean-властивостей використовується PropertyValueFactory, для обчислюваних – лямбда-вирази з CellDataFeatures:

```
// PropertyValueFactory
nameCol.setCellValueFactory(
    new PropertyValueFactory<>("name"));


// Обчислювана властивість через лямбду
avgCol.setCellValueFactory(c ->
    new SimpleDoubleProperty(
        c.getValue().averageGrade()));
```

Для editable-таблиці в адмінуванні текст варіантів редагується через TextFieldTableCell.forTableColumn, а ознака правильності – через CheckBoxTableCell.forTableColumn. Це забезпечує WYSIWYG-редагування безпосередньо в таблиці.

Графіки. Два типи графіків: BarChart<String, Number> для частотного словника у вкладці «Аналіз тексту» та LineChart<String, Number> для динаміки балів у вкладці «Прогрес». Графіки наповнюються через XYChart.Series та XYChart.Data. Анімація вимкнена (animated=false) для стабільності відображення. Осі конфігуруються через CategoryAxis та NumberAxis із мітками та діапазонами.

Зведену статистику обсягу реалізації GUI наведено у таблиці 3.1.

Таблиця 4.1 – Кількісні характеристики реалізації GUI

Показник	Значення
FXML-файлів	10
Загальний обсяг FXML	611 рядків
CSS-файлів	1 (app.css)
Обсяг CSS	187 рядків
Класів-контролерів	10
TableView	6
Графіки (BarChart + LineChart)	2
Стиль-класів у CSS	18
Загальний обсяг Java-коду	3 522 рядків (33 файли)

4.3. Тестування програмного продукту

Тестування навчального тренажера здійснено на трьох рівнях: модульне автоматичне тестування (JUnit 5) [33], ручне функціональне тестування сценаріїв та верифікація виконання вимог (функціональних Ф1–Ф10 та нефункціональних НФ1–НФ8, визначених у розділі 1).

Модульне тестування (JUnit 5) [32]. Автоматичними тестами покрито п'ять ключових класів функційного та сервісного шару – ті, що містять чисту логіку без залежності від JavaFX і FXML. Загалом написано 26 тестових методів у 5 класах (268 рядків). Тести запускаються командою mvn test і виконуються за ~0.5 с.

Перелік тестових класів та охоплених сценаріїв наведено у таблиці 4.2.

Таблиця 4.2 – Перелік модульних тестів та охоплених сценаріїв

Тестовий клас	Кількість тестів	Що перевіряється
StudentsAnalyticsTest	5	filterAndSort: поріг, сортування за спаданням; topByAverage: ліміт; groupByCity: збереження загальної кількості; overallSummary: count = сума оцінок; порожній список
TextAnalyticsTest	6	tokenize: розбиття за пунктуацією; wordCount: підрахунок усіх; uniqueWordCount: підрахунок унікальних; wordFrequencies: сортування за спаданням; порожній текст; longestWord: коректність
PipelineBuilderTest	5	порожній конвеєр = identity; multiply+add vs add+multiply: порядок кроків; apply: незмінність вхідного списку; filterEven: заміна непарних на 0; abs+negate: комбінація

Продовження таблиці 4.2

OptionalDemoTest	7	parseInt: валідне число, невалідне, null, порожній рядок; safeDivide: нормальне ділення, ділення на 0, некоректний ввід; safeLength: null-safe; safeUpperCase: порожній, null, нормальний
QuizServiceTest	3	singleChoice: правильний/неправильний id; multipleChoice: точна множина, підмножина, зайвий варіант; fillIn: нормалізація пробілів і регістру

Розглянемо кілька тестів детальніше.

Тест порядку кроків у конвеєрі перевіряє, що compose(multiply(2), add(1)) дає інший результат, ніж compose(add(1), multiply(2)):

```
@Test
void multiplyThenAdd_isOrderSensitive() {
    var p1 = List.of(
        new Step(MULTIPLY, 2), new Step(ADD, 1));
    var p2 = List.of(
        new Step(ADD, 1), new Step(MULTIPLY, 2));
    assertEquals(7, combine(p1).applyAsInt(3)); // (3*2)+1
    assertEquals(8, combine(p2).applyAsInt(3)); // (3+1)*2
}
```

Цей тест верифікує, що andThen дійсно є некомутативною операцією – ключовий навчальний момент у темі композиції функцій.

Тест нормалізації FILL_IN перевіряє, що різне форматування не впливає на результат:

```
@Test
void fillIn_normalizesWhitespaceAndCase() {
    Question q = new Question();
    q.setType(FILL_IN);
    q.setCorrectAnswer("(a,b)->a+b");
    assertTrue(service.checkAnswer(q, "(a,b) -> a+b"));
    assertTrue(service.checkAnswer(q, "(A, B)->A+B"));
```

```
    assertFalse(service.checkAnswer(q, "a+b"));
}
```

Тест незмінності вхідного списку гарантує, що функційний модуль не модифікує оригінал – принцип чистоти:

```
@Test
void apply_doesNotMutateInput() {
    var input = List.of(1, 2, 3);
    var steps = List.of(new Step(SQUARE, 0));
    var result = PipelineBuilder.apply(input, steps);
    assertEquals(List.of(1, 4, 9), result);
    assertEquals(List.of(1, 2, 3), input); // not changed
}
```

Ручне функціональне тестування. Кожен із п'яти навчальних сценаріїв перевірено на робочій конфігурації Windows 10 / JDK 26 / JavaFX 26.0.1. У таблиці 3.3 наведено протокол ключових перевірок.

Таблиця 4.3 – Протокол ручного функціонального тестування

№	Сценарій / дія	Очікуваний результат	Фактичний результат
1	Вхід: створити профіль "Василь Григоренко" (921)	Профіль створено, перехід до головного вікна	Відповідає
2	Stream API: filter з порогом 75, натиснути filter+sorted	Студенти з avg>=75 у порядку спадання	Відповідає
3	Stream API: натиснути groupingBy(city)	Map за містами з переліком студентів	Відповідає
4	Аналіз тексту: завантажити приклад, аналізувати	Частотний словник, BarChart, 4 показники	Відповідає
5	Конвеєр: додати x*2, додати x>3?x:0, застосувати	Правильний результат конвеєра	Відповідає
6	Optional: parseInt("abc")	Optional.empty()	Відповідає
7	Optional: safeDivide("5", "0")	Optional.empty()	Відповідає
8	Тренажер: обрати всі теми, 10 питань, пройти	Питання перемішані, бал обчислено	Відповідає

Продовження таблиці 4.3

9	Тренажер: MULTIPLE_CHOICE – обрати підмножину	Невірно (потрібна точна множина)	Відповідає
10	Прогрес: перевірити після тренування	Спроба з'явилась у журналі, графік оновлено	Відповідає
11	Експорт CSV	Файл збережено через FileChooser	Відповідає
12	Адмінування: створити нове питання FILL_IN	Питання збережено, з'явилося у списку	Відповідає
13	Теорія: обрати тему Optional	Відображено форматований текст із кодом	Відповідає
14	Вийти → повторний вхід іншим профілем	Коректна зміна профілю, прогрес окремий	Відповідає

Верифікація функціональних вимог. У таблиці 4.4 наведено зведену перевірку виконання кожної з десяти функціональних вимог, визначених у розділі 1.

Таблиця 4.4 – Верифікація виконання функціональних вимог

Вимога	Зміст	Статус
Ф1	Stream API на колекції студентів (CRUD + 7 операцій + код)	Виконано
Ф2	Аналіз тексту (токенізація, частотний словник, BarChart, показники)	Виконано
Ф3	Конструктор конвеєра (9 операцій, andThen, статистика)	Виконано
Ф4	Optional (parseInt, safeDivide, safeLength, orElseGet)	Виконано
Ф5	Тренажер (3 типи питань, перемішування, пояснення)	Виконано
Ф6	Профілі користувачів (створення, вибір, зберігання у БД)	Виконано
Ф7	Облік прогресу (журнал, статистика, LineChart)	Виконано
Ф8	Теоретичний довідник (5 тем, TextFlow)	Виконано
Ф9	Експорт CSV (Stream + Collectors.joining)	Виконано
Ф10	Адмінування банку питань (CRUD, валідація, editable таблиця)	Виконано

Верифікація нефункціональних вимог. У таблиці 4.5 зведено перевірку нефункціональних вимог.

Таблиця 4.5 – Верифікація виконання нефункціональних вимог

Вимога	Зміст	Спосіб перевірки	Статус
НФ1	Зручність	Ручне тестування UI, відгуки	Виконано
НФ2	Локальність	Робота без Інтернету після завантаження	Виконано
НФ3	Кросплатформенність	Java + JavaFX (перевірено на Windows)	Виконано
НФ4	Продуктивність (<200 мс)	Візуальна оцінка часу відгуку	Виконано
НФ5	Надійність	Некоректний ввід, порожні поля, дублікати	Виконано
НФ6	Розширюваність	Додавання питань через адмін-вкладку	Виконано
НФ7	Тестованість	26 JUnit-тестів проходять успішно	Виконано
НФ8	Відповідність ФП	Код модулів використовує Stream, Optional, andThen	Виконано

Аналіз покриття тестами. Модульними тестами покрито всі п'ять класів функційного та сервісного шару (StudentsAnalytics, TextAnalytics, PipelineBuilder, OptionalDemo, QuizService). Ці класи є «ядром» навчальної логіки: саме вони виконують обчислення, перетворення та перевірку. GUI-контролери [34] не покриті автоматичними тестами, оскільки їх тестування потребує JavaFX Application Thread та графічного середовища, що виходить за межі JUnit. Для контролерів використано ручне функціональне тестування (таблиці 3.3).

Середовище тестування. Ручне тестування проведено на конфігурації: Windows 10, JDK 26 (Java(TM) SE Runtime Environment build 26.0.1+8-34), JavaFX 26.0.1, SQLite 3.46.1, Apache Maven 3.9, Microsoft VS Code 1.97. Компіляція 33 Java-файлів виконується без помилок та попереджень [35]. Запуск через mvn javafx:run стартує програму за ~2 с (після першого завантаження залежностей). Усі 26 модульних тестів проходять зі статусом SUCCESS.

ВИСНОВКИ

У бакалаврській роботі спроєктовано та програмно реалізовано настільний навчальний тренажер з теми «Функційне програмування в Java» дисципліни «Сучасні парадигми програмування». Усі поставлені задачі виконано в повному обсязі.

1. Проаналізовано функційне програмування як парадигму та визначено шість ключових концепцій, що відображені у тренажері: чисті функції, незмінність даних, функції вищого порядку, композиція функцій, декларативна обробка колекцій, безпечна робота з відсутніми значеннями через Optional. Для кожної концепції визначено засоби реалізації в Java SE (лямбда-вирази, функціональні інтерфейси java.util.function, Stream API, Collectors, Optional, andThen/compose).
2. Виконано огляд існуючих навчальних засобів (інтерактивні платформи, документація, IDE-інструменти, спеціалізовані застосунки) та виявлено прогалину – відсутність локального тренажера, що системно охоплює тему ФП у Java з контролем знань і фіксацією прогресу.
3. Сформульовано 10 функціональних (Ф1–Ф10) та 8 нефункціональних (НФ1–НФ8) вимог до тренажера. Обґрунтовано вибір технологічного стека: Java 21 LTS, JavaFX 26, SQLite 3.46, Maven, JUnit 5. Спроєктовано трирівневу архітектуру з п'ятьма пакетами (model, db, dao, service/functional, ui), що забезпечує чітке розмежування відповідальностей та незалежне тестування кожного шару.
4. Алгоритмізовано п'ять навчальних сценаріїв: демонстрація Stream API (7 операцій на колекції студентів), аналіз тексту (токенізація, частотний словник, гістограма), конструктор конвеєра (композиція через andThen із 9 типами операцій), демонстрація Optional (4 патерни), контроль знань (3 типи питань: SINGLE_CHOICE, MULTIPLE_CHOICE, FILL_IN із перемішуванням). Побудовано блок-схеми ключових алгоритмів.

5. Спроєктовано реляційну модель даних із 6 таблиць (users, topics, questions, answer_options, attempts, attempt_answers) з каскадним видаленням та 3 індексами. Спроєктовано графічний інтерфейс: 2 екрани (вхід, головне вікно), 8 вкладок (Stream API, Аналіз тексту, Конвеєр функцій, Optional, Тренажер, Прогрес, Теорія, Адмінування), 10 FXML-розміток із CSS-стилізацією.
6. Програмно реалізовано 33 Java-класи (3 522 рядки коду), 10 FXML-розміток (611 рядків), CSS-файл стилів (187 рядків). Чотири функціональні модулі (StudentsAnalytics, TextAnalytics, PipelineBuilder, OptionalDemo) реалізовано як чисті функції без побічних ефектів. Підсистема тренажера (QuizService) забезпечує формування набору питань, перевірку відповідей трьох типів, фіксацію спроб у БД. Банк даних містить 32 питання за п'ятьма темами.
7. Тестування проведено на трьох рівнях. Модульне тестування: 26 тестів JUnit 5 у 5 класах (StudentsAnalyticsTest, TextAnalyticsTest, PipelineBuilderTest, OptionalDemoTest, QuizServiceTest) – усі проходять зі статусом SUCCESS. Ручне функціональне тестування: 14 перевірок ключових сценаріїв на конфігурації Windows 10 / JDK 26 / JavaFX 26.0.1 – усі відповідають. Верифікація вимог: 10 функціональних та 8 нефункціональних вимог виконано.

Напрями подальшого розвитку: розширення банку питань до 100+ (у тому числі параметризовані питання з генерованим кодом); додавання режиму «навчання з помилок» (повторення питань, на які студент відповів невірно); інтеграція з LMS (Moodle) для централізованого збору результатів; реалізація паралельних стрімів як окремого модуля; портування на мобільну платформу (Android) або веб (JavaFX → React).

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Van Roy P., Haridi S. Concepts, Techniques, and Models of Computer Programming. Cambridge, MA : MIT Press, 2004. 900 p. URL: https://www.info.ucl.ac.be/~pvr/book.html (дата звернення: 04.05.2026).
2. Hudak P., Hughes J., Peyton Jones S., Wadler P. A History of Haskell: Being Lazy with Class. Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III). New York : Association for Computing Machinery, 2007. P. 12-1–12-55. DOI: 10.1145/1238844.1238856. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/history.pdf (дата звернення: 04.05.2026).
3. Java Platform, Standard Edition 21 API Specification. Oracle Corporation, 2024. URL: https://docs.oracle.com/en/java/javase/21/docs/api/index.html (дата звернення: 04.05.2026).
4. Goetz B. State of the Lambda. OpenJDK Project, 2013. URL: https://cr.openjdk.org/~briangoetz/lambda/lambda-state-final.html (дата звернення: 04.05.2026).
5. Hughes J. Why Functional Programming Matters. Research Topics in Functional Programming / ed. D. Turner. Reading, MA : Addison-Wesley, 1990. P. 17–42. URL: https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf (дата звернення: 04.05.2026).
6. Backus J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM. 1978. Vol. 21, No. 8. P. 613–641. DOI: 10.1145/359576.359579. URL: https://dl.acm.org/doi/pdf/10.1145/359576.359579 (дата звернення: 04.05.2026).
7. Wadler P. Monads for Functional Programming. Advanced Functional Programming. Lecture Notes in Computer Science. Vol. 925. Berlin :

Springer, 1995. P. 24–52. URL: https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf (дата звернення: 04.05.2026).

8. Church A. An Unsolvable Problem of Elementary Number Theory. American Journal of Mathematics. 1936. Vol. 58, No. 2. P. 345–363. DOI: 10.2307/2371045. URL: https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/church.pdf (дата звернення: 04.05.2026).
9. Java 8 Stream API Tutorial. Baeldung. URL: https://www.baeldung.com/java-8-streams (дата звернення: 04.05.2026).
10. JavaFX 21 Documentation. OpenJFX. URL: https://openjfx.io/javadoc/21/ (дата звернення: 04.05.2026).
11. SQLite Documentation. SQLite Consortium. URL: https://www.sqlite.org/docs.html (дата звернення: 04.05.2026).
12. Goetz B. Translation of Lambda Expressions. OpenJDK Project, 2012. URL: https://cr.openjdk.org/~briangoetz/lambda/lambda-translation.html (дата звернення: 04.05.2026).
13. Lambda Expressions. The Java Tutorials. Oracle Corporation. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html (дата звернення: 04.05.2026).
14. Gosling J., Joy B., Steele G., Bracha G., Buckley A., Smith D. B. The Java Language Specification, Java SE 21 Edition. Oracle Corporation, 2023. URL: https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf (дата звернення: 04.05.2026).
15. Package java.util.stream. Java Platform SE 21 API Specification. Oracle Corporation. URL: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/stream/package-summary.html (дата звернення: 04.05.2026).
16. Lindholm T., Yellin F., Bracha G., Buckley A., Smith D. B. The Java Virtual Machine Specification, Java SE 21 Edition. Oracle Corporation, 2023. URL:

https://docs.oracle.com/javase/specs/jvms/se21/jvms21.pdf (дата звернення: 04.05.2026).

17. Processing Data with Java SE 8 Streams. Oracle Technology Network, 2014. URL: https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html (дата звернення: 04.05.2026).
18. Guide to Java Optional. Baeldung. URL: https://www.baeldung.com/java-optional (дата звернення: 04.05.2026).
19. Odersky M., Rompf T. Unifying Functional and Object-Oriented Programming with Scala. Communications of the ACM. 2014. Vol. 57, No. 4. P. 76–86. DOI: 10.1145/2591013. URL: https://dl.acm.org/doi/pdf/10.1145/2591013 (дата звернення: 04.05.2026).
20. Ierusalimschy R., de Figueiredo L. H., Celes W. Passing a Language through the Eye of a Needle. ACM Queue. 2011. Vol. 9, No. 5. P. 20–29. DOI: 10.1145/1978862.1983083. URL: https://dl.acm.org/doi/pdf/10.1145/1978862.1983083 (дата звернення: 04.05.2026).
21. Claessen K., Hughes J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000). ACM, 2000. P. 268–279. DOI: 10.1145/351240.351266. URL: https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quick.pdf (дата звернення: 04.05.2026).
22. Peyton Jones S. Haskell 98 Language and Libraries: The Revised Report. Journal of Functional Programming. 2010. Vol. 13, No. 1. P. 1–255. URL: https://www.haskell.org/onlinereport/haskell2010/ (дата звернення: 04.05.2026).
23. Getting Started with JavaFX. OpenJFX. URL: https://openjfx.io/openjfx-docs/ (дата звернення: 04.05.2026).

24. Introduction to FXML. JavaFX Documentation. Oracle Corporation. URL: https://docs.oracle.com/javase/8/javafx/api/javafx/fxml/doc-files/introduction_to_fxml.html (дата звернення: 04.05.2026).
25. JavaFX CSS Reference Guide. OpenJFX, 2024. URL: https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/doc-files/cssref.html (дата звернення: 04.05.2026).
26. Memon A. M. An Event-Flow Model of GUI-Based Applications for Testing. Software Testing, Verification and Reliability. 2007. Vol. 17, No. 3. P. 137–157. DOI: 10.1002/stvr.364. URL: https://cs.umd.edu/~atif/papers/MemonSTVR2007.pdf (дата звернення: 04.05.2026).
27. JUnit 5 User Guide. Version 5.10.2. JUnit Team, 2024. URL: https://junit.org/junit5/docs/5.10.2/user-guide/ (дата звернення: 04.05.2026).
28. Apache Maven Project. Getting Started Guide. Apache Software Foundation. URL: https://maven.apache.org/guides/getting-started/index.html (дата звернення: 04.05.2026).
29. JSR 335: Lambda Expressions for the Java Programming Language. Java Community Process, 2014. URL: https://jcp.org/en/jsr/detail?id=335 (дата звернення: 04.05.2026).
30. SQLite Foreign Key Support. SQLite Consortium. URL: https://www.sqlite.org/foreignkeys.html (дата звернення: 04.05.2026).
31. Owens M. The Definitive Guide to SQLite. 2nd ed. Berkeley, CA : Apress, 2010. 368 p. URL: https://link.springer.com/book/10.1007/978-1-4302-3226-1 (дата звернення: 04.05.2026).
32. xerial/sqlite-jdbc: SQLite JDBC Driver. GitHub Repository. URL: https://github.com/xerial/sqlite-jdbc (дата звернення: 04.05.2026).
33. Pears A., Seidman S., Malmi L., Mannila L., Adams E., Bennedsen J., Devlin M., Paterson J. A Survey of Literature on the Teaching of Introductory Programming. ACM SIGCSE Bulletin. 2007. Vol. 39, No. 4. P. 204–223. DOI: 10.1145/1345375.1345441. URL:

https://dl.acm.org/doi/pdf/10.1145/1345375.1345441 (дата звернення: 04.05.2026).

34. Syme D., Granicz A., Cisternino A. Expert F# 4.0. 4th ed. Berkeley, CA : Apress, 2015. 638 p. URL: https://link.springer.com/book/10.1007/978-1-4842-0740-6 (дата звернення: 04.05.2026).
35. Milner R., Tofte M., Harper R., MacQueen D. The Definition of Standard ML, Revised. Cambridge, MA : MIT Press, 1997. 128 p. URL: https://smlfamily.github.io/sml97-defn.pdf (дата звернення: 04.05.2026).
36. Krasner G. E., Pope S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming. 1988. Vol. 1, No. 3. P. 26–49. URL: https://www.lri.fr/~mbl/ENS/FONDIHM/2013/papers/Krasner-JOOP88.pdf (дата звернення: 04.05.2026).
37. JEP 395: Records. OpenJDK, 2020. URL: https://openjdk.org/jeps/395 (дата звернення: 04.05.2026).
38. JEP 394: Pattern Matching for instanceof. OpenJDK, 2021. URL: https://openjdk.org/jeps/394 (дата звернення: 04.05.2026).
39. JEP 361: Switch Expressions. OpenJDK, 2020. URL: https://openjdk.org/jeps/361 (дата звернення: 04.05.2026).
40. Lloyd J. W. Practical Advantages of Declarative Programming. Proc. Joint Conference on Declarative Programming (GULP-PRODE 1994). 1994. P. 3–17. URL: https://cgi.cse.unsw.edu.au/~eptcs/Published/GULP-PRODE94/Proceedings/lloyd.pdf (дата звернення: 04.05.2026).
41. Abelson H., Sussman G. J. Structure and Interpretation of Computer Programs. 2nd ed. Cambridge, MA : MIT Press, 1996. 657 p. URL: https://web.mit.edu/6.001/6.037/sicp.pdf (дата звернення: 04.05.2026).
42. Fowler M. Collection Pipeline. martinfowler.com, 2015. URL: https://martinfowler.com/articles/collection-pipeline/ (дата звернення: 04.05.2026).

43. Brusilovsky P., Miller P. Course Delivery Systems for the Virtual University. Access to Knowledge: New Information Technologies and the Emergence of the Virtual University / ed. F. T. Tschang, T. Della Senta. Amsterdam : Elsevier, 2001. P. 167–206. URL: https://www.pitt.edu/~peterb/papers/VirtUniv.pdf (дата звернення: 04.05.2026).
44. The Stream API. Dev.java – Official Java Developer Portal. Oracle Corporation. URL: https://dev.java/learn/api/streams/stream-api/ (дата звернення: 04.05.2026).
45. Karavirta V., Helminen J., Ihantola P. A Mobile Learning Application for Parsons Problems with Automatic Feedback. Proc. 12th Koli Calling International Conference on Computing Education Research. ACM, 2012. P. 11–18. DOI: 10.1145/2401796.2401798. URL: https://dl.acm.org/doi/pdf/10.1145/2401796.2401798 (дата звернення: 04.05.2026).

ДОДАТКИ

Додаток А «Лістинг коду LoginController.java»

```
package com.example.fpjava.ui;

import com.example.fpjava.dao.UserDao;
import com.example.fpjava.model.User;
import javafx.fxml.FXML;
import javafx.scene.control.Alert;
import javafx.scene.control.Button;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.control.Tooltip;

import java.util.function.Consumer;

public class LoginController {

    @FXML private TextField nameField;
    @FXML private TextField groupField;
    @FXML private ChoiceBox<User> usersChoice;
    @FXML private Button loginExistingBtn;
    @FXML private Button createBtn;
    @FXML private Label statusLabel;

    private final UserDao userDao = new UserDao();
    private Consumer<User> onLoggedIn;

    public void setOnLoggedIn(Consumer<User> onLoggedIn) {
        this.onLoggedIn = onLoggedIn;
    }
```

```
    @FXML
    public void initialize() {
        reloadUsers();
        loginExistingBtn.setTooltip(new Tooltip("Увійти під обраним
профілем"));
        createBtn.setTooltip(new Tooltip("Створити новий профіль"));
    }

    private void reloadUsers() {
        usersChoice.getItems().setAll(userDao.findAll());
        if (!usersChoice.getItems().isEmpty()) {
            usersChoice.getSelectionModel().selectFirst();
        }
    }

    @FXML
    private void onCreate() {
        String name = nameField.getText() == null ? "" :
nameField.getText().trim();
        String group = groupField.getText() == null ? "" :
groupField.getText().trim();
        if (name.isBlank()) {
            statusLabel.setText("Вкажіть прізвище та ім'я");
            return;
        }
        try {
            User u = userDao.create(name, group);
            statusLabel.setText("Профіль створено: " + u.getName());
            reloadUsers();
            usersChoice.getSelectionModel().select(u);
            if (onLoggedIn != null) onLoggedIn.accept(u);
        } catch (Exception e) {
            showError("Не вдалося створити профіль: " +
e.getMessage());
        }
```

```
    }

    @FXML
    private void onLoginExisting() {
        User u = usersChoice.getValue();
        if (u == null) {
            statusLabel.setText("Оберіть існуючий профіль або
створіть новий");
            return;
        }
        if (onLoggedIn != null) onLoggedIn.accept(u);
    }

    private void showError(String msg) {
        Alert a = new Alert(Alert.AlertType.ERROR, msg);
        a.setHeaderText(null);
        a.showAndWait();
    }
}
```

Додаток Б «Лістинг коду StudentsAnalytics.java»

```
package com.example.fpjava.functional;

import com.example.fpjava.model.Student;

import java.util.Arrays;
import java.util.Comparator;
import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

/**
 * Демонстрація операцій Stream API на колекції студентів.
 * Усі методи є чистими функціями: не модифікують вхідні дані,
повертають нові колекції.
 */
public final class StudentsAnalytics {

    private StudentsAnalytics() {
    }

    /** Базовий тестовий набір студентів. */
    public static List<Student> sampleStudents() {
        return Arrays.asList(
                new Student("Іваненко І.І.", "Полтава", "ПМ-21",
List.of(85, 90, 78, 92, 88)),
                new Student("Петренко О.С.", "Київ",    "ПМ-21",
List.of(70, 65, 72, 68, 75)),
                new Student("Сидоренко М.В.", "Полтава", "ПМ-22",
List.of(95, 98, 92, 97, 94)),
                new Student("Коваленко А.Б.", "Львів",   "ПМ-21",
List.of(60, 58, 62, 55, 65)),
```

```
                new Student("Шевченко Д.К.", "Київ",     "ПМ-22",
List.of(80, 82, 85, 78, 88)),
                new Student("Бондаренко Р.Г.", "Харків",  "ПМ-22",
List.of(72, 75, 70, 73, 78)),
                new Student("Мельник Т.О.", "Полтава", "ПМ-21",
List.of(90, 92, 88, 94, 91)),
                new Student("Ткаченко В.П.", "Львів",    "ПМ-22",
List.of(50, 55, 48, 52, 58))
        );
    }

    /** Студенти із середнім балом ≥ threshold, відсортовані за
зменшенням середнього. */
    public static List<Student> filterAndSortByAverage(List<Student>
students, double threshold) {
        return students.stream()
                .filter(s -> s.averageGrade() >= threshold)

.sorted(Comparator.comparingDouble(Student::averageGrade).reversed()
)
                .collect(Collectors.toList());
    }

    /** Топ-N студентів за середнім балом. */
    public static List<Student> topByAverage(List<Student> students,
int limit) {
        return students.stream()

.sorted(Comparator.comparingDouble(Student::averageGrade).reversed()
)
                .limit(limit)
                .collect(Collectors.toList());
    }
```

```
    /** Групування студентів за містом. Зберігаємо порядок ключів
через LinkedHashMap. */
    public static Map<String, List<Student>>
groupByCity(List<Student> students) {
        return students.stream()
                .collect(Collectors.groupingBy(
                        Student::getCity,
                        LinkedHashMap::new,
                        Collectors.toList()));
    }

    /** Групування студентів за навчальною групою. */
    public static Map<String, List<Student>>
groupByGroup(List<Student> students) {
        return students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGroup,
                        LinkedHashMap::new,
                        Collectors.toList()));
    }

    /** Середній бал у розрізі міст. */
    public static Map<String, Double> averageByCity(List<Student>
students) {
        return students.stream()
                .collect(Collectors.groupingBy(
                        Student::getCity,
                        LinkedHashMap::new,

Collectors.averagingDouble(Student::averageGrade)));
    }

    /** Кількість студентів у кожній групі. */
    public static Map<String, Long> countByGroup(List<Student>
students) {
```

```
        return students.stream()
                .collect(Collectors.groupingBy(
                        Student::getGroup,
                        LinkedHashMap::new,
                        Collectors.counting()));
    }

    /** Розгорнутий список усіх оцінок усіх студентів через flatMap.
*/
    public static List<Integer> allGrades(List<Student> students) {
        return students.stream()
                .flatMap(s -> s.getGrades().stream())
                .collect(Collectors.toList());
    }

    /** Агрегована статистика по всіх оцінках. */
    public static IntSummary overallSummary(List<Student> students)
{
        var stats = students.stream()
                .flatMapToInt(s ->
s.getGrades().stream().mapToInt(Integer::intValue))
                .summaryStatistics();
        return new IntSummary(stats.getCount(), stats.getMin(),
stats.getMax(), stats.getAverage(), stats.getSum());
    }

    /** Узагальнений запис для статистики по цілих числах. */
    public record IntSummary(long count, int min, int max, double
average, long sum) {
        @Override
        public String toString() {
            return String.format("count=%d, min=%d, max=%d,
avg=%.2f, sum=%d",
                    count, min, max, average, sum);
        }
```

```
}
```