

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування**

На правах рукопису
УДК _____

До захисту допущено
Завідувач кафедри

_____ Вадим МУХІН
«__» _____ 2024 р.

**Магістерська дисертація
на здобуття ступеня магістра
за освітньо-професійною програмою
“Інтелектуальні сервіс-орієнтовані розподілені обчислювання”
зі спеціальності 122 "Комп'ютерні науки"**

**на тему: «Застосування предметно-орієнтованих мов програмування для
розробки Android-застосунків»**

Виконав:
студент II курсу, групи ДА-21мп
Токар Михайло Євгенович _____

Керівник:
доцент, к.т.н.,
Булах Богдан Вікторович _____

Рецензент:
к.т.н., доц., зав. каф. ММСА НН ІПСА
Тимошук Оксана Леонідівна _____

Засвідчую, що у цій магістерській
дисертації немає запозичень з праць
інших авторів без відповідних
посилань

Студент (*підпис*): _____

Київ – 2024

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

**Навчально-науковий інститут прикладного системного аналізу
Кафедра системного проектування**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма – «Інтелектуальні сервіс-орієнтовані розподілені обчислювання»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Вадим МУХІН**

25 червня 2023 р.

**ЗАВДАННЯ
на магістерську дисертацію студенту**

Токар Михайлу Євгеновичу

1. Тема дисертації «Застосування предметно-орієнтованих мов програмування для розробки Android-застосунків», науковий керівник дисертації Булах Богдан Вікторович, к.т.н., доц. каф. СП, затверджені наказом по університету від 08 листопада 2023 року № 5200-с

2. Термін подання студентом дисертації – 8 січня 2023 р.

3. Об'єкт дослідження

Сучасні методології розробки предметно-орієнтованих розширень мов програмування (domain-specific language, DSL), призначені для створення клієнтських застосунків на платформі Android, інструментарій, що використовується у процесі створення зазначених розширень, а також методологічні підходи та критерії оцінки, спрямовані на визначення їхньої ефективності, придатності та ергономіки у контексті сучасних вимог до мобільної розробки.

4. Вихідні дані

Опис існуючих рішень та інструментарію в області розробки предметно-орієнтованих розширень мов програмування на платформі Android, а також інші інструменти та фреймворки, що застосовуються в даній галузі.

5. Перелік завдань, які потрібно розробити

1. Теоретичні засади предметно-орієнтованих мов програмування.
2. Сучасні інструментальні засоби для розробки DSL.
3. Застосування DSL у програмуванні для платформи Android.
4. Реалізація програмних прототипів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу

1. Презентація до захисту роботи.

7. Орієнтовний перелік публікацій

1. Тези доповіді на науково-практичній конференції.

8. Дата видачі завдання – 25 червня 2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Початок практики	01 вересня 2023	
2	Ознайомлення з нормативними документами	07 вересня 2023	
3	Теоретичні засади предметно-орієнтованих мов програмування	15 вересня 2023	
4	Сучасні інструментальні засоби для розробки DSL	15 жовтня 2023	
5	Застосування DSL у програмуванні для платформи Android	25 листопада 2023	
6	Програмна реалізація програмних прототипів	5 грудня 2023	
7	Висновки щодо проведеного дослідження.	15 грудня 2023	
8	Підготовка дисертації до захисту	31 грудня 2023	
9	Захист дисертації	8 січня 2024	

Студент
Науковий керівник

Михайло ТОКАР
Богдан БУЛАХ

АНОТАЦІЯ

Магістерська дисертація містить 131 с., 25 табл., 23 рис., 1 додаток, 83 джерела.

Актуальність теми: тема є актуальною завдяки постійно зростаючому попиту на гнучкі, адаптивні та ефективні підходи до розробки мобільного програмного забезпечення.

Об'єкт дослідження: розробка та застосування предметно-орієнтованих мов для програмування додатків на платформі Android.

Предмет дослідження: сучасні методології та інструментарій для розробки, а також аналіз ефективності застосування DSL у контексті мобільної розробки на платформі Android.

Мета дослідження: розробка та аналіз ефективності застосування DSL у контексті Android-розробки.

Постановка задачі: аналіз та дослідження підходів та інструментів для розробки DSL та їх адаптація до потреб Android-розробки, програмна реалізація різних підходів та аналіз ефективності створених рішень

За допомогою парсер-генератора ANTLR та можливостей мови програмування Kotlin було створено відповідно зовнішню та внутрішню DSL, проведено порівняння та аналіз ефективності кожного з підходів.

Ключові слова: DSL, Kotlin, Android, ANTLR, предметно-орієнтована мова, граматика, синтаксис, інтерпритатор, абстрактне синтаксичне дерево.

ABSTRACT

The master`s thesis contains 131 p., 25 tabl., 23 fig., 1 appendix, 83 ref.

Relevance of the topic: the topic is relevant due to the ever-growing demand for flexible, adaptive and efficient approaches to mobile software development.

Object of research: development and application of object-oriented languages for programming applications on the Android platform.

Subject of research: modern methodologies and tools for development, as well as analysis of the effectiveness of DSL in the context of mobile development on the Android platform.

The purpose of the research: to develop and analyze the effectiveness of DSL in the context of Android development.

Statement of the problem: to analyze and study approaches and tools for DSL development and adapt them to the needs of Android development, to implement different approaches in software and to analyze the effectiveness of the created solutions

Using the ANTLR parser-generator and the capabilities of the Kotlin programming language, an external and internal DSL were created, respectively, than the effectiveness of each approach was compared and analyzed.

Keywords: DSL, Kotlin, Android, ANTLR, domain-specific language, grammar, syntax, interpreter, abstract syntax tree.

ЗМІСТ

ЗМІСТ	6
ВСТУП	7
1 ТЕОРЕТИЧНІ ЗАСАДИ ПРЕДМЕТНО-ОРІЄНТОВАНИХ МОВ ПРОГРАМУВАННЯ	9
1.1 Аналіз основних понять та класифікацій мов програмування	10
1.2 Дослідження проектування та побудови предметно-орієнтованих мов програмування	16
1.3 Висновки до розділу	25
2 СУЧАСНІ ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ДЛЯ РОЗРОБКИ DSL	26
2.1 Текстові мови	26
2.2 Графічні мови	30
2.3 Проекційні редактори	33
2.4 Можливості мови Kotlin	35
2.5 Висновки до розділу	40
3 ЗАСТОСУВАННЯ DSL У ПРОГРАМУВАННІ ДЛЯ ПЛАТФОРМИ ANDROID	42
3.1 Використання внутрішніх DSL	43
3.2 Використання зовнішніх DSL	62
3.3 Висновки до розділу	69
4 ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОГРАМНИХ ПРОТОТИПІВ	72
4.1 Програмний прототип зовнішньої DSL	74
4.2 Програмний прототип внутрішньої DSL	77
4.3 Висновки до розділу	78
5 РОЗРОБКА СТАРТАП ПРОЕКТУ	83
5.1 Розробка плану стартапу та його ринкове масштабування	83
5.2 Опис ідеї стартап-проекту	84
5.3 Технологічний аудит ідеї проекту	85
5.4 Ринкові можливості для запуску стартап-проекту	87
5.5 Розробка ринкової стратегії стартап-проекту	92
5.6 Розробка маркетингової програми стартап-проекту	93
5.7 Висновки до розділу	94
ВИСНОВКИ	96
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	98
ДОДАТКИ	106

ВСТУП

У сучасному світі, де технології розвиваються стрімко, важливість мобільних застосунків та їх ефективної розробки набуває особливого значення. Одним із ключових напрямків у цій сфері є застосування предметно-орієнтованих мов програмування (DSL) для розробки Android-застосунків. Ця магістерська дисертація зосереджується на дослідженні та розробці DSL для платформи Android.

Актуальність даної теми обумовлена постійно зростаючим попитом на гнучкі, адаптивні та ефективні підходи до розробки мобільного програмного забезпечення.

Мета даної роботи полягає у розробці та аналізі ефективності застосування DSL у контексті Android-розробки. Це включає дослідження сучасних методологій, інструментарію та практичного застосування DSL, а також розробку та порівняння програмних прототипів на основі обраних підходів.

Для досягнення поставленої мети були визначені наступні **завдання**:

1. Аналіз підходів, методів та алгоритмів, що використовуються у сфері розробки предметно-орієнтованих мов (DSL) з акцентом на їх застосування для платформи Android.
2. Дослідження сучасних інструментів для розробки DSL та їх адаптації до потреб Android-платформи.
3. Розробка програмних прототипів з використанням різних DSL підходів.
4. Тестування та аналіз ефективності розроблених програмних рішень.

Об'єктом дослідження є сучасні методології розробки предметно-орієнтованих мов для програмування та їх застосування додатків на платформі Android.

Предметом дослідження є інструментарій для створення DSL-розширень, а також методологічні підходи та критерії оцінки ефективності, придатності та ергономіки DSL у контексті мобільної розробки на платформі Android.

Апробація результатів дослідження включає порівняльний аналіз, архітектурний огляд, тестування та верифікацію розроблених програмних прототипів, а також оцінку їх придатності для реальних застосунків.

Структура роботи відображає логічну послідовність дослідження та складається з п'яти основних розділів, які охоплюють теоретичні засади DSL, аналіз інструментарію для їх розробки, практичне застосування DSL у програмуванні для Android, та програмну реалізацію, порівняння й тестування розроблених прототипів, а також розробку стартап проекту за визначеною тематикою.

1 ТЕОРЕТИЧНІ ЗАСАДИ ПРЕДМЕТНО-ОРІЄНТОВАНИХ МОВ ПРОГРАМУВАННЯ

Даний розділ присвячено дослідженню сутності та ключових методологій, що стосуються предметно-орієнтованих мов програмування (DSL) та аналізу контексту їх застосування в розробці клієнтських застосунків для операційної системи Android.

Перед нами стоїть завдання не просто описати та оглянути DSL і пов'язаних з ним концепцій, але й провести комплексний аналіз еволюції, теоретичних засад та методологічних підходів. Варто зазначити, що в контексті еволюції DSL ми також розглядатимемо процес адаптації мов, інструментарію та суміжних методологій до динамічно змінюваних вимог ринку та технологічного середовища, а також оцінимо їх вплив на ефективність розробки мобільних додатків.

Також даний розділ допоможе нам зрозуміти, як саме DSL можуть бути використані для створення більш ефективних, гнучких та функціональних мобільних застосунків.

Ми розпочнемо з огляду основних понять та класифікації DSL, розглянемо різні типи, їх застосування та процес еволюції. Далі ми зосередимося на аналізі методів та алгоритмів, які лежать в основі розробки DSL, з акцентом на їх застосування в контексті Android. Це дозволить нам не лише зрозуміти технічні аспекти DSL, але й оцінити їх вплив на загальний процес розробки мобільних застосунків.

Таким чином, цей розділ стане фундаментом для глибшого розуміння предметно-орієнтованих мов програмування, а також їх застосування у світі Android-розробки, що є критично важливим для досягнення цілей нашої роботи.

1.1 Аналіз основних понять та класифікацій мов програмування

Перед оглядом наукових думок щодо визначення предметно-орієнтованих мов програмування слід надати базис того, що взагалі таке мова програмування і як ми визначаємо її.

У 1966 році, Американській інститут стандартів визначає наступне формулювання: мова програмування – мова що використовується для підготовки комп'ютерних програм [1]. Нідерландський словник обробки інформації трактує загальний термін “мова” наступним чином: мова – загальний термін для позначення певного набору символів і правил або конвенцій, що регулюють спосіб і послідовність, у якій ці символи можуть бути об'єднані для змістовної комунікації. З поміткою “Однозначна мова, призначена для вираження програм, називається *мовою програмування*” [2]. Автор мови C++ Б'ярн Струструп надає більш прикладний набір характеристик, котрий визначає що таке мова програмування. На його думку, *мова програмування* – це концепція, котра поєднує в собі наступні характеристики: інструмент для навчання машин, засіб спілкування між програмістами, засіб для вираження високорівневих проектів, нотація для алгоритмів, спосіб вираження взаємозв'язків між концепціями, інструмент для експериментування, засіб керування комп'ютеризованими пристроями [3]. Італійські дослідники Мауріціо Габбріеллі та Сімоне Мартіні надають наступне визначення: мова програмування - це штучний формалізм, за допомогою якого можна виражати алгоритми, котрий незважаючи на всю свою штучність, залишається мовою [4]. Особливо цікаво, що в контексті їх дослідження мова програмування розглядається в першу чергу як лінгвістичний конструкт, котрий в загальному лінгвістичному випадку має наступні характеристики для дослідження: синтаксис, семантика та прагматика. У випадку ж мови програмування на думку авторів варто розрізняти ще одну характеристику: реалізація.

Узагальнюючи все вищеописане, на думку автора коректно сформульоване визначення в контексті даної роботи звучить наступним чином:

Мова програмування — це лінгвістична система символів та правил, що визначаються синтаксисом, семантикою, граматикою та реалізацією і використовується для створення інструкцій керування поведінкою обчислювальної машини, а також служить як засіб комунікації між програмістами та комп'ютерними системами, дозволяючи формувати логічні послідовності для вирішення різноманітних задач.

Визначення коректного для нашої роботи терміну “мова програмування” веде нас до огляду класифікаційних ознак мов програмування. Під час огляду літератури, найбільш коректне визначення ознаки класифікації, котрі ми знайшли надали В.М. Олійник, котрий визначає ознаку класифікації як властивість об'єкта класифікованої множини, котра може мати кількісне або якісне значення [5] та Г.М. Рябенко, котрий визначає даний термін як спільні риси, притаманні системі підпорядкованих понять (класів) [6]. Класифікація є критично необхідним методом для подальшого аналізу теоретичних засад необхідних нам для досягнення завдань роботи. Використання даного підходу дозволить нам глибше зрозуміти різноманіття, специфіку та категоризацію мов програмування. Проаналізувавши наявну літературу, нами було визначено наступний перелік класифікаційних ознак мов програмування:

1. Рівень абстракції.
2. Підтримка парадигм програмування.
3. Способи реалізації мов.
4. Тип семантики.
5. Застосування

В контексті комп'ютерних наук, абстракція означає створення моделі об'єкта чи явища, яка ігнорує менш важливі деталі, зосереджуючись на ключових аспектах. Рівень абстракції, таким чином, визначає глибину, з якою ми досліджуємо систему чи процес, дозволяючи нам вибірково зосереджуватися на

найважливіших характеристиках. Це є основоположним для розуміння, проектування та реалізації комп'ютерних систем, програмного забезпечення та алгоритмів [7].

Для ілюстрації можна проаналізувати концепцію рівня абстракції у сфері програмування. У цьому контексті рівень абстракції визначається як ступінь відокремлення деталей реалізації програми від її функціональності. Наприклад, високорівневі мови програмування, такі як Python чи Java, надають більш абстрактний підхід до програмування, що дозволяє розробникам зосередитися на логіці та функціональності програми, не вдаючись у деталі, такі як управління пам'яттю чи архітектура процесора.

У відношенні до архітектури комп'ютерних систем, рівень абстракції може вказувати на різні шари системи, починаючи від апаратного забезпечення і закінчуючи операційною системою та прикладними програмами. Наприклад, операційна система виступає як абстракція, яка приховує складність апаратного забезпечення від кінцевих користувачів та програмістів.

Ми визначаємо рівень абстракції у комп'ютерних науках як концептуальний інструмент, що дозволяє розглядати комп'ютерні системи, програмне забезпечення та алгоритми на різних ступенях деталізації, відокремлюючи важливі аспекти від менш значущих. Це сприяє зосередженню розробників та архітекторів систем на вищих рівнях логіки та функціональності, мінімізуючи необхідність уваги до низькорівневих деталей і спрощуючи процеси проектування, розробки та управління комп'ютерними системами.

Підтримка парадигми програмування в рамках мов програмування визначається здатністю мови надавати конструкції, ідіоми та механізми, які сприяють використанню програмістами певного стилю чи підходу до програмування. Парадигма програмування не обмежується лише синтаксисом чи правилами мови; вона також визначає філософію, що лежить в основі процесу розробки програмного забезпечення.

Парадигма програмування – це концептуально узгоджений підхід до структурування програм, який визначається способом використання концепцій та методів в рамках програмного проекту [8].

Об'єктно-орієнтоване програмування (ООП) у сфері програмної інженерії представляє собою парадигму, що базується на концепції об'єктної декомпозиції задач. Вона описує структуру програмного забезпечення через взаємодію об'єктів, а поведінку системи через механізм обміну повідомленнями між цими об'єктами. Ключові аспекти цієї парадигми включають інкапсуляцію даних та поведінки, наслідування, поліморфізм та класифікацію об'єктів. Програмні мови, які підтримують ООП, такі як Java, C++ та Python, надають відповідні конструкції для реалізації цих концепцій [8].

Функціональне програмування є парадигмою програмування, яка базується на концептуальному принципі математичних функцій, акцентує увагу на незмінності стану та відсутності побічних ефектів у процесі виконання програми. Відмінна від імперативного стилю програмування, функціональне програмування орієнтоване на декларативне визначення взаємодії через функції, зокрема, використання функцій вищого порядку та рекурсивних структур. Мови, які підтримують цю парадигму, включають Haskell, Scala та інші, що включають механізми для ефективного управління незмінними даними та функціональних патернів [9].

Процедурне програмування є парадигмою програмування, що використовує методологію структурованого виклику процедур або підпрограм для виконання задач. Процедурне програмування орієнтоване на лінійну організацію коду через визначені процедури та підпрограми, що дозволяє чітко структурувати програмний код. Мови, які підтримують цю парадигму, такі як C і Pascal, пропонують засоби для створення чітко організованого програмного коду з використанням структурованих блоків [9].

Логічне програмування є частиною дискретної математики та парадигмою програмування, що базується на використанні логічних висновків. Програма у

цій парадигмі розглядається як набір фактів та правил, що дозволяє виводити нові факти на основі вже існуючих. Мова Prolog є яскравим представником цієї парадигми, де програми складаються з логічних висловлювань і правил для виведення нової інформації [10].

Важливість підтримки програмувальних парадигм мовами програмування має значний вплив на методологію розробки програмного забезпечення. Ця підтримка диктує методи, якими розробники можуть підходити до вирішення програмних завдань. Кожна парадигма пропонує унікальні переваги, що відображається у різних аспектах розробки. Об'єктно-орієнтоване програмування сприяє створенню модульного та легко перевикористовуваного коду, завдяки її структурі, заснованій на об'єктах та класах. Функціональне програмування може значно покращити читабельність та надійність коду, особливо у контексті багатопотокових та паралельних обчислень. Процедурне програмування ефективно застосовується у сценаріях, де потрібен детальний контроль над структурою та логікою виконання програми. Логічне програмування виявляється особливо корисним у сферах, пов'язаних із штучним інтелектом та обробкою даних, де потрібно використання складних логічних висновків [10].

У рамках теорії та практики програмування, розгляд різних типів мов програмування, таких як компільовані, інтерпретовані та гібридні, відіграє критичну роль у виборі інструментів для реалізації програмного забезпечення. Кожен з цих типів має унікальні характеристики, які впливають на процес розробки та виконання програм.

Компіляція вихідного коду в машинний код, яка є характерною для компільованих мов, забезпечує високу продуктивність виконання та глибокий статичний аналіз. Однак, це також включає часові затрати на компіляцію та певну залежність від апаратної платформи [8].

Інтерпретовані Мови використовують інтерпретатор для безпосереднього виконання вихідного коду. Це забезпечує високу гнучкість, зручність у

використанні, ідеально підходячи для швидкого розроблення та скриптів. Проте, такий підхід може призвести до зниження продуктивності виконання.

Гібридні мови об'єднують елементи компільованих та інтерпретованих підходів, компілюючи код у проміжний формат, що потім інтерпретується або додатково компілюється. Такий підхід покликаний забезпечити баланс між швидкістю розробки та оптимізацією продуктивності, з використанням проміжного байт-коду для забезпечення платформи-незалежності.

Ці фундаментальні відмінності між підходами до реалізації мов програмування відіграють вирішальну роль у виборі технологій для розробки програмного забезпечення, впливаючи на ключові аспекти, такі як ефективність, гнучкість, та портативність коду. Вибір конкретної мови та підходу до реалізації залежить від унікальних вимог конкретного проекту програмного забезпечення та переваг розробників.

У рамках теоретичного аналізу мов програмування, виокремлюються два головні напрями відносно семантики: статична семантика та динамічна семантика. Ці напрямки відіграють значущу роль у процесах аналізу та виконання програм, володіючи різноманітними унікальними властивостями та областями застосування [8].

Статична семантика зосереджується на аспектах програми, які піддаються аналізу на стадії компіляції, тобто перед її виконанням. Вона охоплює елементи такі як перевірка типів, область видимості змінних та інші фактори, які можна визначити без безпосереднього запуску коду. Часто включає в себе строгу систему типів, що дозволяє ідентифікувати помилки на ранніх стадіях розробки, ще до виконання програми. А статичний аналіз дозволяє компілятору оптимізувати код, використовуючи доступну інформацію для генерації більш ефективного машинного коду [8].

Динамічна семантика відображає поведінку програми під час її фактичного виконання. Вона охоплює аспекти, такі як способи виконання інструкцій, управління пам'яттю, обробка виключень, та інші властивості, що стають релевантними в процесі виконання [8].

У рамках систематичного аналізу мов програмування, їх класифікація за призначенням та областями застосування має вирішальне значення для визначення адекватності вибору мови для специфічних проектів та завдань. Дві основні категорії в цій класифікації включають мови програмування загального призначення (GPL) та предметно-орієнтовані мови (DSL). GPL характеризуються широким спектром застосувань та високою адаптивністю до різних областей програмування, включаючи веб-розробку, системне програмування та інше. GPL мови, такі як Java, C++, та Python, забезпечують багатий функціонал та розширені бібліотеки, що дозволяє розробникам ефективно вирішувати широкий спектр завдань, пропонують значну варіативність у підходах та методиках для реалізації різноманітних проектів, а застосування цих мов охоплює широкий спектр програмних продуктів від простих скриптів до складних системних рішень [9].

DSL спроектовані ж з урахуванням конкретних областей застосування або специфічних завдань. Вони пропонують високий рівень абстракції та спеціалізовані інструменти для певних сфер, таких як веб-розробка, наукове обчислення, або управління базами даних. DSL мови, як от SQL для баз даних або HTML для веб-розробки, надають інструменти, спеціально призначені для конкретних завдань.

Вибір між GPL та DSL мовами заснований на аналізі конкретних потреб проекту та задач, які потрібно вирішити, з урахуванням специфіки області застосування та вимог до функціональності та ефективності програмного забезпечення.

1.2 Дослідження проектування та побудови предметно-орієнтованих мов програмування

Історія предметно-орієнтованих мов (DSL) тісно переплетена з еволюцією обчислювальної техніки. У період раннього розвитку комп'ютерних наук, коли основна увага приділялась створенню універсальних мов програмування, вже були зроблені спроби створити спеціалізовані мови для конкретних доменів.

Основна мета цих мов полягала у забезпеченні більш ефективного та зручного способу вирішення специфічних завдань у певних галузях. З часом, як технології розвивалися, DSL почали набувати ширшого поширення, пропонуючи рішення для комплексних задач, таких як обробка даних, управління системами та специфічні бізнес-процеси.

Сучасний етап розвитку DSL характеризується інтенсивним використанням у різних областях, від медичних досліджень до веб-розробки. Ці мови не лише дозволяють розробникам більш ефективно вирішувати доменно-специфічні задачі, а й сприяють підвищенню якості та надійності розроблюваних рішень. Історія та розвиток DSL - це історія пошуку оптимальних шляхів для вирішення унікальних проблем, що вимагають нестандартних підходів. Даний підрозділ присвячено широкому огляду історії, розвитку та еволюції предметно-орієнтованих мов.

DSL — це спеціальні мови програмування, розроблені для певної предметної області. Вони стають все більш популярними в галузі інженерії програмного забезпечення, оскільки можуть підвищити продуктивність, якість та ефективність розробки програмного забезпечення. DSL використовуються в багатьох методологіях розробки програмного забезпечення, таких як генеративне програмування, продуктові лінії, програмні фабрики, мовно-орієнтоване програмування та модельно-орієнтоване проектування. Незважаючи на свою популярність, дослідження DSL поки що не стали самостійною галуззю досліджень зі сформованими дослідницькими групами та довготривалими конференціями/семінарами. В широкому сенсі предметно-орієнтовані мови існують далеко поза межами комп'ютерних наук. Виходячи з нашого трактування предметно-орієнтованих мов, такими можуть вважатись наприклад нотний нотація, азбука морзе, схема в'язки, тощо.

Для досягнення цілей нашого дослідження, нам необхідно провести теоретичний аналіз ключових понять та аспектів, що застосовуються в даній сфері. Сформулюємо визначення того, що взагалі таке предметно-орієнтована мова.

Почнемо з визначення наданого Л. Беттіні, який визначає предметно-орієнтовані мови, як мови програмування або мови специфікацій, орієнтовані на певну проблемну область та не призначені для того, щоб надавати можливості для розв'язання всіх типів проблем, а також, якщо область вашої проблеми охоплюється певною DSL, вона може й має розв'язати цю проблему простіше і швидше замість GPL [11].

Група науковців очолювана Т. Косаром надає наступне визначення: предметно-орієнтована мова (DSL) — це мова призначена для забезпечення нотації, пристосованої до прикладної області, і базується лише на відповідних поняттях та особливостях цієї предметної області. Таким чином, DSL є засобом опису та генерації членів сімейства програм в межах певної предметної області, без необхідності знань про загальне програмування [12].

Маркус Фоельтер [13] визначає предметно-орієнтовані мови як ті, що оптимізовані для певного класу задач, котрий ми називаємо називаємо доменом (предметом). Вони базується на абстракціях, які тісно пов'язані з доменом, для яких вони створені. Ці мови також мають синтаксис, придатний для стислого вираження цих абстракцій. У більшості випадках це текстові позначення, але таблиці, символи (як у математиці) або графіки також можуть бути корисними. За умови добре визначеної визначеної семантики цих абстракцій ефективність вираження програм для спеціалізованої області значно зростає.

Схожу думку розділяє й Дебасіш Гош, наводячі наступне визначення: DSL – це мова програмування, орієнтована на вирішення конкретної проблеми, визначаючи синтаксис і семантику, які моделюють концепції на тому ж рівні абстракції, який пропонує проблемна область [14].

В ході нашого дослідження ми можемо виокремити одну з популярних класифікаційних ознак предметно-орієнтованих мов програмування – класифікацією за типом інтеграції.

Таку думку підтримує М. Фуллер, котрий розділяє DSL на зовнішні (external) та внутрішні (internal) предметно-орієнтовані мови [15]: Зовнішні DSL Фуллер трактує як специфічні для домену мови, представлені окремою мовою,

відмінною від основної мови програмування, з якою вона працює. Ця мова може використовувати власний синтаксис, а може слідувати синтаксису іншого представлення, наприклад, XML. Внутрішню ж DSL на його думку, представлена у синтаксисі мови загального призначення. Це стилізоване використання цієї мови для специфічних для домену цілей.

Термін же “вбудована DSL” в контексті синоніму до “зовнішньої DSL”, хоч і широко застосований, Фоулер рекомендує уникати, оскільки "вбудована мова" може також застосовуватися до мов сценаріїв, вбудованих у додатки, таких як VBA в Excel або Scheme в Gimp, що має набагато більшу конкретику, ніж зовнішня предметно-орієнтована мова.

Схожу думку розділяє й М. Фельтер, визначаючи внутрішні предметно-орієнтовані мови (Internal DSLs) як синтаксичний конструкт, що природньо інтегруються в мови загального призначення [13]. Основа таких DSL часто заснована на метапрограмуванні, що є характерним для динамічно типізованих мов, але Scala становить виняток, будучи статично типізованою мовою з виведенням типів. Особливість вбудованих DSL полягає в тонкій межі між ними та API, особливо коли йдеться про Fluent API, які характеризуються ланцюговими викликами методів, формуючи більш концентрований код. Вбудовані DSL (Internal DSLs) відрізняються від Fluent API своєю глибиною інтеграції та обсягом можливостей в мові програмування. Внутрішні DSL інтегруються безпосередньо у мову програмування, пропонуючи більш багатий та гнучкий синтаксис, тоді як Fluent API є більш обмеженими, оскільки вони являють собою лише ланцюжок викликів методів, який виглядає більш читабельно і організовано. Fluent API не змінюють основного синтаксису хост-мови, в той час як внутрішні DSL можуть надавати ілюзію окремої, спеціалізованої мови. Цей підхід дозволяє створювати послідовності викликів, подібні до граматики, що надає Fluent API рис DSL.

Фельтман вказує на обмеження вбудованих DSL, зокрема, відсутність інтеграції з інтегрованими середовищами розробки (IDE), що є критично важливою для їх прийняття та використання. Він наголошує, що особливо це

стосується динамічно типізованих мов таких як Python, де IDE не може повноцінно підтримувати граматику та обмеження вбудованого DSL [16].

Цю ж думку розділя С. Гюнтера існує два типи DSL: зовнішні та внутрішні. Він визначає зовнішні DSL, як ті що вимагають створення мови з нуля, з власним синтаксисом і семантикою, що відбувається за рахунок реалізації необхідних компіляторів/інтерпретаторів та інших інструментів самостійно або за допомогою середовища розробки мови. Внутрішні DSL, за його інтерпритацією навпаки, будуються на основі вже існуючої мови програмування, яка також називається хост-мова [16].

Деякі науковці, як от група очолювана Томашем Косаром [12], проводячи своє дослідження не розділяють предметно орієнтовані мови на внутрішні та зовнішні, натомість наводячи більш широку класифікацію, котра виглядає наступним чином:

1. Попередня обробка (Preprocessing) – переклад конструкцій DSL у конструкції базової мови, обмежуючи статичний аналіз до того, що виконує процесор базової мови. До цього методу відносяться такі підшаблони як макрообробка, де нові конструкції визначаються через інші конструкції мови; трансформація з коду на код, де код DSL перекладається у код існуючої мови; пайплайн, що включає послідовну обробку підмов DSL.
2. Вбудовування (Embedding): Використання існуючих механізмів базової мови для побудови бібліотеки операцій, специфічних для домену. Такий підхід дозволяє виразити ідіому домену за допомогою синтаксичних механізмів базової мови.
3. Розширюваний компілятор/інтерпретатор (Extensible compiler/interpreter): Розширення існуючого компілятора або інтерпретатора за допомогою специфічних для домену правил оптимізації та генерації коду, з використанням можливостей, як-от рефлексія.
4. Компілятор/Інтерпретатор (Compiler/Interpreter): Використання стандартних технік компілятора або інтерпретатора для реалізації DSL,

включаючи переклад конструкцій DSL у конструкції базової мови і повний статичний аналіз програми або специфікації DSL.

5. Генератор компіляторів (Compiler generator): Цей підхід схожий на компілятор/інтерпретатор, але деякі фази реалізовані з використанням систем розробки мови або інструментів для написання компіляторів, мінімізуючи зусилля на реалізацію.

Узагальнюючі, можемо сказати, що перші 3 техніки: попередня обробка, вбудовування та розширення компілятора/інтерпретатора ми можемо класифікувати як інструменти притаманні внутрішнім предметно-орієнтованим мовам, а 4 та 5 техніку, котрі пов'язані з створенням нового компілятора до технік створення зовнішніх DSL.

Подібне широке визначення наводить також Ар'є ван Деурсен [17], визначаючи наступні типи реалізації предметно-орієнтованих мов:

1. Попередня обробка або макрообробка: Техніка, яка перекладає нові DSL конструкції у вирази базової мови за допомогою препроцесора, зосереджуючись на простоті реалізації.
2. Розширення базової мови: Метод, що полягає в доповненні існуючої мови програмування специфічними для домену конструкціями, зберігаючи при цьому основні функції базової мови.
3. Вбудовані мови / бібліотеки для конкретних доменів: Підхід, що використовує існуючі механізми хост-мови для створення бібліотеки операцій, специфічних для домену.
4. Інтерпретація або компіляція: Класичний підхід до створення нової мови з використанням стандартних або спеціалізованих інструментів для DSL, що забезпечує повну адаптацію під конкретний DSL.
5. Розширюваний компілятор або інтерпретатор: Метод, що інтегрує фазу попередньої обробки у компілятор, що покращує перевірку типів та оптимізацію.

В подібному розділенні ми бачимо подібне до попереднього розділення, аналогічно можемо віднести перші три підходи до внутрішніх DSL, а 4 та 5 – до зовнішніх.

Варто також зазначити, що деякі науковці, як от Ф. Томасетті [18], вважають дане розділення недоцільним, а термін внутрішня предметно-орієнтована мова некоректним, декларуючи що дані конструкції є всього лише програмами, котрі пишуться програмістами для програмістів, що не відповідає ідеології предметно-орієнтованих мов і не може називатись такою, оскільки концепт DSL орієнтуються на широку групу людей в конкретному домені.

В контексті даної роботи ми не погоджуємось з такою думкою, оскільки наше дослідження показало, що в більшості випадків предметно-орієнтована мова, не використовується широким загальним користувачів, а створюється під проблеми не просто конкретного домену, а конкретної компанії в даному домені, та є лише внутрішньою закритою системою, котрі використовуються для зручності співробітниками компанії: від програмістів до працівників технічної підтримки. Тому в цьому контексті ми не вважаємо, що термін внутрішня DSL є некоректним.

Проаналізувавши відповідну літературу, ми надалі в цій роботі визначимо наступні термінологію щодо внутрішніх та зовнішніх предметно-орієнтованих мов:

Внутрішні DSL – мови програмування, що реалізуються на основі вже існуючих мов програмування (хост-мов) або прямо інтегрується в них. Вони використовують синтаксис і семантику цієї мови, але розширюють її набір конструкцій для задоволення специфічних потреб певного домену. Реалізація внутрішніх DSL може бути здійснена за допомогою перекладу конструкцій DSL у конструкції базової мови, використання існуючих механізмів базової мови для побудови бібліотеки операцій, специфічних для домену або ж за допомогою розширення існуючого компілятора або інтерпретатора за допомогою специфічних для домену правил оптимізації та генерації коду.

Зовнішні DSL – мови програмування, що реалізуються окремо, як незалежні мови, не використовуючи синтаксис і семантику базової мови, натомість декларуючі свої унікальні синтаксичні конструкти. Реалізація зовнішніх DSL вимагає створення мови з нуля, з власним компілятором/інтерпретатором та іншими допоміжними інструментами.

Іншу класифікацію вирізняє Маркус Фюельтер [13] поділяючи предметно-орієнтовані мови на технічні та прикладні. Автор зазначає, що перший тип використовуються програмістами, а другий – непрограмістами, що важливо враховувати при проектуванні проектування DSL. Ми погоджуємося з даною класифікацією, виокремлюючи окрему класифікаційну ознаку – за цільовими користувачами.

В ході аналізу літературних джерел [14, 13, 15, 19] нами було визначено розділення на три типи предметно-орієнтованих мов: текстові мови, графічні мови, проєкційні редактори, більш детально на кожному типі ми зупинимось в наступних розділах.

Проекційний редактор дозволяє користувачеві ефективно редагувати представлення коду у вигляді абстрактного синтаксичного дерева (АСТ). Він може імітувати поведінку текстового редактора для текстових нотацій, редактора діаграм для графічних мов, табличного редактора для редагування таблиць тощо.

У контексті інформатики та теорії формальних мов, ключове значення має процес синтаксичного аналізу, відомий як парсинг [20]. Цей процес полягає у розборі граматичної структури вхідної послідовності символів згідно із заданою формальною граматикою. Синтаксичний аналіз включає два основних етапи: ідентифікацію осмислених токенів через лексичний аналіз та створення синтаксичного дерева, яке відображає структуру вхідних даних за допомогою парсера [21].

Формальна граматика є фундаментальним поняттям у теорії формальних мов, що описує формальну мову через виділення підмножини з множини всіх слів скінченного алфавіту. Ця концепція була введена Ноамом Чомскі у 1950-х

роках. Важливим інструментом у визначенні формальних граматики є нотація Бекуса-Наура (BNF), яка використовується для опису контекстно-вільних граматики, типових для мов програмування та протоколів комунікації [20].

У рамках синтаксичного аналізу, використовуються різні види синтаксичних дерев, серед яких конкретне синтаксичне дерево (також відоме як дерево розбору) та абстрактне синтаксичне дерево (AST). Конкретне синтаксичне дерево є впорядкованим деревом, в якому корінь позначений аксіомою граматики, проміжні вершини містять нетермінали, а листя – термінали [22].

Абстрактне синтаксичне дерево (AST), з іншого боку, відображає структуру програми вищого рівня, ігноруючи певні синтаксичні елементи, які не впливають на семантику програми. В AST внутрішні вершини асоційовані з операторами мови програмування, а листя - з відповідними операндами. AST використовується для проміжного представлення програми між конкретним синтаксичним деревом та структурою даних, що використовується в компіляторах чи інтерпретаторах для оптимізації та генерації коду [20].

Важливим елементом синтаксичного аналізу є використання різних типів аналізаторів, зокрема LL-аналізаторів. LL-аналізatori обробляють вхідні дані зліва направо, будуючи ліворекурсивне виведення рядка. Вони можуть бути класифіковані як LL(k)-аналізatori, де 'k' вказує на кількість токенів, що розглядаються аналізатором під час обробки виразу. Такі аналізатори не обмежені скінченною кількістю токенів для попереднього перегляду та можуть використовувати регулярні вирази для прийняття рішень [23].

Іншою фазою обробки коду під час компіляції є лексичний аналіз. Лексичні аналізатори сегментують вхідний рядок у синтаксичні одиниці, звані лексемами, та класифікують їх у відповідні класи токенів. Ці аналізатори можуть бути простими або складними, залежно від необхідності обробки структури фрази та взаємодії з парсером [24].

У поняття токена вбудовується концепція лексеми, яка є одиницею мови та включає в себе всі парадигматичні форми одного слова та їхні лексичні

значення. Токен в свою чергу є об'єктом, утвореним з лексеми під час лексичного аналізу, а шаблон токена слугує як формальний опис класу лексем, що можуть утворювати цей тип токена [25].

Загалом, процес синтаксичного аналізу та пов'язані з ним компоненти, включаючи формальну граматику, синтаксичні дерева, аналізатори та лексичний аналіз, є невід'ємною частиною обробки та розуміння мов програмування, що сприяє ефективному перекладу та виконанню програмного коду.

1.3 Висновки до розділу

В даному розділі нашого дослідження, нами було проаналізовано основні поняття та класифікацію мов програмування, а також проведено дослідження щодо теоретичних засад проектування та побудови різноманітних мов програмування, зокрема й предметно-орієнтованих.

В ході аналізу нами було визначено класифікаційні ознаки мов програмування, та проаналізовану кожен з наступних класифікацій: рівень абстракції, підтримка парадигмі програмування, способи реалізації мов, тип семантики та застосування. Саме за застосуванням можна виокремити категорію предметно-орієнтованих мов, від інших мов програмування, що є виживим фактом в нашому дослідженні.

Також нами було визначено розділення на три типи предметно-орієнтованих мов: текстові мови, графічні мови, проєкційні редактори, що є важливою характеристикою при проектуванні мови.

Було розглянута теоретична основа для створення власної мови: у теорії формальних мов, ключове значення має процес синтаксичного аналізу, відомий як парсинг. Цей процес полягає у розборі граматичної структури вхідної послідовності символів згідно із заданою формальною граматиною. Розглянуто складові побудови власної мови програмування, як ось: формальна граматика, парсер, лексер, абстрактне синтаксичне дерево, синтаксичний та лексичний аналізи, токен тощо.

2 СУЧАСНІ ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ДЛЯ РОЗРОБКИ DSL

Існують різні способи побудови предметно-орієнтованих мов програмування. Мета такої розробки – створити мову з інструментальною підтримкою, не докладаючи при цьому надмірних зусиль [18]. В контексті створення своєї мови програмування для певного домену, перед розробника не стоїть задача створити наступну Java або C#, тому вимоги до користування цією мовою не співставні з мільйонами людино годин, що витрачаються під час розробки GPL на створення надскладного компілятора або інтегрованого середовища розробки з безліччю можливостей. Серед першочергових задач в розробці DSL стоїть створення корисної мови, з хорошою інструментальною підтримкою, котру може підтримувати невелика команда науковців або розробників.

2.1 Текстові мови

Розглянемо інструментарій для створення текстових мов. В широкому використанні найбільш класичні мови. Більшість практиків навіть не уявляють собі інші види мов. Деякі вчені, як от Ф. Томасеті зазначають, що текстові мови легше підтримувати і їх можна використовувати в будь-яких контекстах, оскільки більшість користувачів звикли працювати з текстовими мовами. Однак, Томасеті також зазначає, що для продуктивного використання таким мовам обов'язково необхідний спеціальний редактор [26]. Розглянемо процес розробки таких мов детальніше.

Проаналізувавши професійну та наукову літературу на дану тематику, ми визначали низку інструментів, що застосовуються для створення текстових мов.

Одним з таких інструментів є ANTLR. ANTLR (ANother Tool for Language Recognition) – це потужний генератор синтаксичних аналізаторів для читання, обробки, виконання або перекладу структурованого тексту або двійкових файлів. Він широко використовується для створення мов, інструментів та фреймворків.

На основі граматики ANTLR генерує синтаксичний аналізатор, який може будувати дерева синтаксичного розбору і ходити ними [27, 28].

ANTLR фактично є сучасним аналогом зв'язки синтаксичного аналізатора yacc та лексичного аналізатора lex. На початку 1970-х років Стівен Джонсон – комп'ютерний вчений з Bell Labs of AT&T, розробив синтаксичний аналізатор Yacc, тому що він хотів вставити оператор XOR у компілятор мови програмування B. Відповідно до цього завдання Спочатку Yacc був написаний мовою програмування B, але незабаром був переписаний Аланом Снайдером на C та згодом з'явився як частина версії 3 Unix [29] а повний опис Yacc був опублікований у 1975 році [30].

Lex – в свою чергу, це комп'ютерна програма, яка генерує лексичні аналізатори [31, 32] та зазвичай використовується з генератором синтаксичного аналізатора yacc. Lex читає вхідний потік, що визначає лексичний аналізатор, і пише вихідний код, який реалізує лексичний аналізатор на мові програмування C [33]. Lex, спочатку написаний Майком Леском та Еріком Шмідтом і описаний у 1975 році все в тій же Bell Labs [34] та є стандартним генератором лексичних аналізаторів у багатьох Unix-системах. Для yacc&lex визначено як частину стандарту POSIX та описану у IEEE POSIX P1003.2, що описує функціональні можливості та вимоги до цих інструментів [35].

Хоча зв'язка інструментів lex & yacc вважається застарілою, а також є пропрієтарним програмним забезпеченням їх згодом почали замінити програмні рішення з відкритим вихідним кодом, котрі є повністю сумісними з оригінальними lex & yacc. Найпопулярнішою зв'язкою таких рішень-замінників були Berkeley Yacc, Flex (fast lexical analyzer) та GNU BISON [31, 32]. Найбільш сучасною й популярною версією, що об'єднує lex та yacc є PLY – реалізація інструментів синтаксичного аналізу lex та yacc для Python, котра початково була розроблена у 2001 році для використання студентами у курсі "Вступ до компіляторів" [36].

ANTLR в свою чергу – це дві речі: інструмент, який перекладає вашу граматику на синтаксичний аналізатор/лексикон на Java (або іншій доступній

цільовій мові), і середовище виконання, необхідне згенерованим синтаксичним аналізаторам/лексиконам. Користувачем визначається граMATика лексера і синтаксичного аналізатора за допомогою ANTLR, а згодом перетворюється дерево розбору, створене ANTLR, у формат, з яким легше працювати. Таким чином отримується модель коду. За допомогою даного інструменту можливо вирішувати питання посилань, будувати валідацію і реалізувати систему типів як набір операцій над моделлю вашого коду, після чого користувач або інтерпретує, або компілює модель коду.

Найбільшим плюсом даного підходу є те, що користувач має повний контроль над тим як будується система. ANTLR має можливості для зміни всієї системи та її подальшого розвитку, що також завдяки зручній документації та великою кількістю прикладів та навіть готових граMATик стає значно простішою задачею, ніж наприклад робота над lex&yacc. Звісно, з таким підходом ми не отримаємо надскладну IDE з десятками операцій рефакторингу, проте це і не входить до задач, що зазвичай ставиться перед розробниками предметно-орієнтованих мов.

ANTLR має дуже широке застосування та використовується наприклад в розробці таких середовищ розробки як IntelliJ IDEA та Xcode [37]. Також варто зазначити, що головним ідеологом та розробником проекту є Теренс Парр – професор комп'ютерних наук з Університету Сан-Франциско. ANTLR на відміну від lex&yacc – це проект з відкритим кодом, починаючи з версії 3.0, який поширюється за ліцензією BSD [28]. Що не менш важливо для розробки мови за допомогою ANTLR існують спеціальні плагіни для інтегрованих середовищ розробки, таких як IntelliJ IDEA чи Visual Studio Code [38, 39], що дещо спрощує розробку та редагування рішень, зроблених за допомогою ANTLR.

Наступним рішенням, що ми розглянемо є Xtext. Eclipse Xtext – це фреймворк для розробки мов програмування та в тому числі предметно-орієнтованих мов. За допомогою Xtext користувач має можливість визначати свою мову за допомогою потужної граMATики. В результаті користувач отримує повну інфраструктуру, включаючи синтаксичний аналізатор, компоновщик,

перевірку типів, компілятор, а також підтримку редагування в Eclipse чи будь-якому редакторі, що підтримує протокол Language Server Protocol [40]

Фактично, Xtext надає інструментарій не лише синтаксичного аналізатора, а й має можливість генерації окремого середовища розробки, котрим можна користуватись як плагіном для Eclipse, або навіть в браузері. На перший погляд дана особливість є беззаперечним плюсом у порівнянні з вищезгаданим ANTLR, що надає лише функціональність парсера, проте для повноцінного використання даної технології необхідно оволодіти не тільки навичками необхідними для написання власної мови, а й вміло використовувати Eclipse Modeling Framework (EMF) – фреймворк моделювання та засіб генерації коду для створення інструментів та інших додатків на основі структурованої моделі даних. На основі специфікації моделі, описаної у XML, EMF надає інструменти та підтримку під час виконання для створення набору класів Java для моделі, а також набір класів-адаптерів, які дозволяють переглядати та редагувати модель на основі команд, а також вищезгаданий базовий редактор [41]. Варто також зазначити, що завдяки широким можливостям даного програмного рішення, користувач має можливість використовувати будь-яку обрану цільову мову для компіляції [42].

Провівши аналіз даної екосистеми, ми можемо декларувати, що інфраструктура Xtext, надана екосистемою Eclipse має надзвичайно широкі можливості, серед яких генерація та редагування редактору коду написаної мови у веб-браузері, в середовищі розробки Eclipse, та навіть в сторонніх проектах як ось IntelliJ IDEA за допомогою генерації додаткових плагінів, проте поріг входження в розробку власної мови за допомогою цієї системи є надзвичайно високим враховуючи вимоги до знання не тільки теорія проектування комп'ютерних мов, а й досвіду роботи з екосистемою Eclipse, зокерма Eclipse Modeling Framework [11].

Вартують згадки також проекти textX та spoofax. TextX – це метамова для створення доменно-специфічних мов (DSL) у Python, котра натхненна вищезгаданим Xtext [43]. Головна задача, котру вирішує дане програмне рішення – створення текстової мови у простий спосіб, за допомогою textX користувач

може спроектувати власну мову або створити підтримку для вже існуючих текстових мов чи форматів файлів. З одного опису мови (граматики) `textX` створить синтаксичний аналізатор і мета-модель (так званий абстрактний синтаксис) для цієї мови. `textX` наслідує синтаксис і семантику `Xtext`, але дещо відрізняється, і на 100% реалізований на Python за допомогою синтаксичного аналізатора `Arpeggio PEG`, котрий виключає граматичні неоднозначності, надає необмежений перегляд, та має стиль роботи інтерпретатора [44, 43].

`Spoofax (Spoofax Language Workbench)` в свою чергу – це платформа для розробки текстових (доменних) мов програмування. Платформа складається з наступних компонентів [45]:

1. Мета-мови для високорівневого декларативного визначення мови.
2. Інтерактивне середовище для розробки мов з використанням цих мета-мов.
3. Генератори коду, які створюють синтаксичні аналізатори, засоби перевірки типів, компілятори, інтерпретатори та інші інструменти на основі визначень мови
4. Генерація повнофункціональних плагінів редактора `Eclipse` на основі визначень мов.
5. API для програмного об'єднання компонентів реалізації мови

Основною перевагою `Spoofax` автори зазначають можливість зосередитися на суті визначення мови та ігнорувати несуттєві деталі реалізації.

2.2 Графічні мови

В ході проведеного аналізу, ми можемо заявити, що графічні мови здаються більш доступними, і часто фахівці з предметних областей почуваються з ними більш невимушено, ніж з текстовими мовами та їхнім складним синтаксисом.

Графічні мови вимагають використання спеціальних редакторів, і вони менш гнучкі, ніж текстові мови. Крім того, вони рідше використовуються, ніж текстові мови, а інструменти для створення графічних мов, як правило, менш досконалі.

Зважаючи на проведений аналіз в даній роботі ми наведемо короткий список прикладів без занурення в деталі щодо їх роботи.

Першим рішенням, котре ми розглянемо буде GMF (Graphical Modeling Framework) – набір генеруючих компонентів та інфраструктури сімейства Eclipse для виконання на льоту для розробки графічних редакторів на основі вже знайомого нам з попереднього розділу EMF та GEF [46].

Подібно до Xtext, він базується на EMF для визначення структури даних, а GMF дозволяє вказувати яким чином різні елементи будуть відображатись, та яким чином будуть представлені їхні зв'язки і т.д.

Серед явних мінусів даної системи наму було визначено, що вона практично не має документації, що робить її використання в контексті розробки мови майже неможливим.

Проте варто зазначити, що завдяки можливостям EMF хоча й має великий потенціал та є потужним та гнучким програмним рішенням.

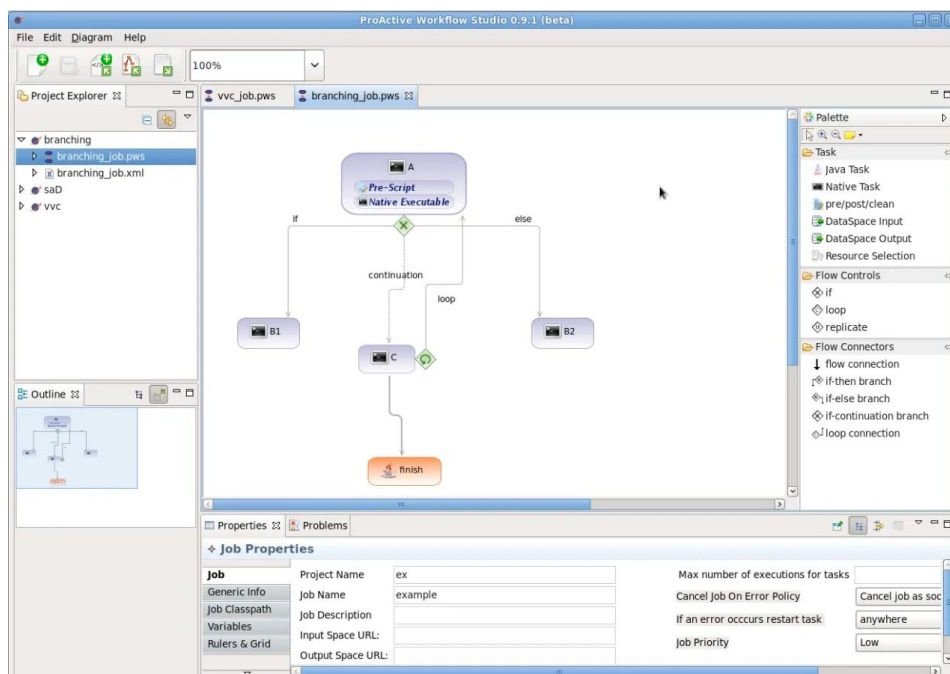


Рисунок 2.1 – Зображення інтерфейсу GMF [47]

Проблеми великого порогу входження до GMF намагаються вирішити декілька сторонніх програмних рішень, що побудовані над GMF – більш простий Eclipse Eugenia, та більш складний Eclipse Sirius. Eclipse Eugenia – це інструмент, який спрощує розробку редакторів графічних моделей на основі GMF,

автоматично генеруючи моделі, необхідні для цього редактора, з однієї анотованої метамоделі визначеної даним програмним продуктом – Ecore [48]. Варто також зазначити, що даний проект наразі перестав підтримуватися на користь більш складного механізму Eclipse Sirius.

Sirius — це проект Eclipse, який дозволяє вам легко створити власний редактор для графічного моделювання, використовуючи технології Eclipse Modeling, включаючи EMF і GMF. Sirius було створено компаніями Obeo та Thales, щоб забезпечити загальну робочу платформу для архітектурного проектування на основі моделей, яку можна було б легко адаптувати до конкретних потреб [49].

Останнім розглянутим нами рішенням в цій царині буде MetaEdit+ – мовне середовище для визначення графічних мов. На відміну від усіх інших інструментів, про які було згадано вище MetaEdit являє собою комерційний інструмент розроблений компанією MetaCase. Компанія MetaCase характеризує MetaEdit+ – як середовище предметно-орієнтованого моделювання, що дозволяє компаніям радикально підвищити продуктивність та якість розробки, генеруючи повний код безпосередньо з моделей. Спочатку створюючи мову моделювання за допомогою MetaEdit+ Workbench, котра впроваджується в роботу компанії, після чого розробники моделюють за допомогою цієї мови в MetaEdit+ Modeler [50].

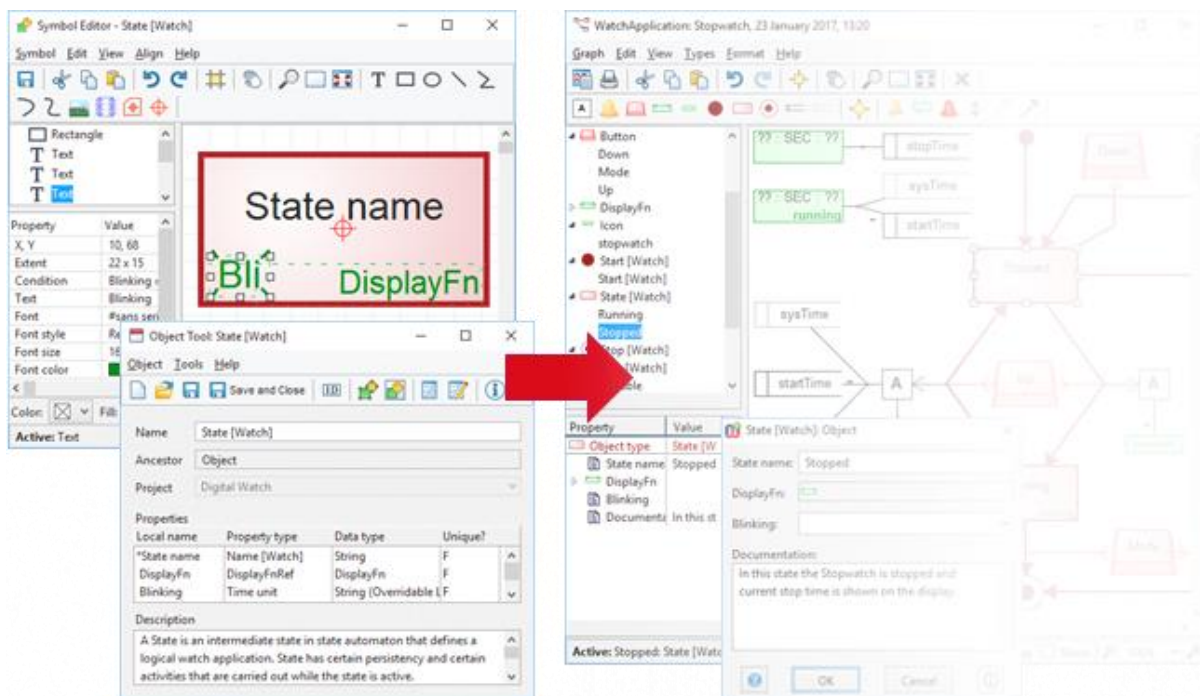


Рисунок 2.2 – Процес розробки та впровадження моделей та моделювання за допомогою MetaEdit+ Workbench [50]

2.3 Проекційні редактори

Перш ніж розглядати саме проекційне редагування ми вважаємо за доцільне описати підхід, що використовується в текстових мовах, в основі якого лежить синтаксичний аналізатор, або ж парсер.

У підході на основі парсера граматики визначає послідовність токенів і слів, що утворюють структурно валідну програму. Парсер генерується з цієї граматики. Парсер – це програма, яка розпізнає валідні програми у текстовій формі та створює абстрактне синтаксичне дерево чи граф. Інструменти аналізу чи генератори працюють з цим абстрактним синтаксичним деревом. Користувачі вводять програми, використовуючи конкретний синтаксис (тобто послідовності символів), і програми також зберігаються в цій формі. Прикладами інструментів у цій категорії є вже вищеописані Spoofax та Xtext [13].

Проекційні редактори (також відомі як структуровані редактори) працюють без граматики та парсерів. Мова визначається шляхом визначення абстрактного синтаксичного дерева, потім визначення правил проекції, які відтворюють конкретний синтаксис мовних концепцій, визначених абстрактним

синтаксисом. Дії редагування безпосередньо змінюють абстрактне синтаксичне дерево. Потім правила проєкції відтворюють текстове (або інше) представлення програми. Користувачі читають та пишуть програми через це проєкційне представлення. Програми зберігаються як абстрактні синтаксичні дерева, зазвичай у форматі XML [13].

Резюмуючи зазначимо, що у підході на основі парсера абстрактне синтаксичне дерево створюється з конкретного синтаксису програми; парсер інстанціює та заповнює абстрактне синтаксичне дерево на основі інформації в тексті програми. У цьому випадку (формальне) визначення (конкретний синтаксис) називається граматикою. Натомість у проєкційному підході абстрактне синтаксичне дерево створюється безпосередньо діями редактора, а конкретний синтаксис відтворюється з цього деревечка через правила проєкції.

На нашу думку JetBrains MPS є найвдалішим прикладом інструменту, який використовує проєкційне редагування, тож саме його ми будемо розглядати в контексті даного дослідження.

JetBrains MPS (The Meta Programming System) – це проєкційний редактор, що як визначено нами з дослідження означає, що в ній не задіяні грамика та синтаксичний аналізатор. Натомість так звані жести редактора змінюють безпосередньо базове абстрактне синтаксичне дерево, яка проєктується так, щоб виглядати як текст. Як наслідок, MPS підтримує змішані нотації (текстові, символні, табличні, графічні) та широкий спектр можливостей компонування мови. MPS має відкритий вихідний код під ліцензією Apache 2.0 і розробляється компанією JetBrains [51].

Серед плюсів даного програмного рішення, котрі зазначає Маркус Фюельтер є підтримка розширених можливостей по редагуванню та створенню своєї мови, не тільки на етапі її проєктування, а й у процесі впровадження у виробничий процес [13].

Отже, проєкційні редактори дають більше можливостей для виразності, оскільки вони не обмежені рамками тексту і синтаксичних аналізаторів, і більше можливостей для інтерактивності, оскільки доступні операції не обмежуються

маніпуляціями з текстом. Часто вони також дозволяють поєднувати різні синтаксиси разом, так що, наприклад, ми можемо вбудовувати діаграми або таблиці, або інші види специфічних для домену представлень, у більш простий текстовий синтаксис.

Зважаючи на все це, звичайно, у проєкційних редакторів є і недоліки. Так наприклад Ф. Томасетті зазначає що вони є надто незвичними для більшості користувачів [26]. Ці типи редакторів існують вже досить давно, і про них є багато літератури. Однак за межами академічних кіл та спеціалізованих ніш проєкційні редактори все ще майже невідомі, і для багатьох користувачів крива навчання може бути досить крутою, особливо якщо ми не витратили багато ресурсів на покращення навчального процесу, як ми вже згадували раніше. Текст є простим, частково через його характеристики, а частково через те, що ми так добре звикли працювати з текстом.

Через цю особливість з'являється низка рішень, котрі дозволяють експортувати проєкційні моделі в текстову репрезентацію, котра є більш звичною розробникам. Одним з таких рішень є LionWeb. LionWeb – метамодельний інструмент метою якого є створення екосистему інтероперабельних компонентів для побудови мовно-орієнтованих інструментів моделювання в Інтернеті [52].

Цей фреймворк дозволяє користувачам експортувати саму мову з MPS до Kotlin, щоб отримати клас Kotlin для кожної моделі-концепту в користувацькій мові. Використовуючи ці згенеровані класи, користувачі можемо створити інтерпретатор на Kotlin.

Це дозволяє користувачам експортувати фактичні стратегії визначені в проєкційному редакторі, написані в MPS за допомогою DSL, у формат LionWeb, щоб ми могли легко прочитати їх з Kotlin і передати нашому інтерпретатору.

2.4 Можливості мови Kotlin

В даному підрозділі ми зосередимося на тому, як Kotlin як сучасна мова програмування, що набула великої популярності у розробці для Android, може

бути використана для створення ефективних та інтуїтивно зрозумілих внутрішніх предметно-орієнтованих мов. Ми розглянемо загальні концепції та принципи, які лежать в основі використання Kotlin для цієї мети, а також проаналізуємо, як ці особливості відкривають нові можливості для розробників у створенні потужних і виразних мовних конструкцій.

При створенні внутрішніх DSL (Domain-Specific Languages) важливою характеристикою мови програмування є її здатність до обмеження та розширення контексту. Внутрішні DSL визначаються своїм контекстом використання, що обумовлює специфічність їхнього застосування у певній предметній області. Можливість обмеження контексту полягає у здатності мови точно виражати лише ті концепції та дії, які є відповідними для даної області, уникненні зайвих абстракцій та деталей загального програмування. Розширення контексту передбачає зручні та ефективні механізми для додавання нових функцій та конструкцій, які можуть бути специфічними для конкретного використання DSL. Ця можливість дає змогу виражати складні концепції в зручній та лаконічній формі, що робить внутрішні DSL потужним інструментом для розробки в різноманітних областях, від програмування веб-додатків до наукових обчислень.

Проаналізувавши низку літературних джерел та відповідну документації мови програмування Kotlin ми знайшли низку особливостей мови програмування Kotlin, що дозволяють створювати, обмежувати та розширювати визначений користувачем контекст [53, 54, 55].

Дані особливості в ході аналізу нами було розділено на два види: користувацькі патерни проектування та вбудовані можливості мови.

До користувацьких патернів, котрі є важливими при проектуванні внутрішньої предметно-орієнтованої мови належать: ланцюжок викликів функцій, контекстні змінні, вкладені конструктори (nested builders) тощо.

Ланцюжок викликів функцій – в даному контексті визначено нами як підхід до програмування, де послідовність функцій викликаються одна за одною,

причому результат виклику однієї функції стає вхідним параметром для наступної.

У ланцюжку викликів функцій кожна функція повертає об'єкт або значення, на якому вона була викликана, що дозволяє послідовно додавати до нього інші методи чи функції. Це створює зручний та читабельний синтаксис, особливо коли потрібно послідовно застосовувати декілька операцій до одного об'єкту.

Ланцюжок викликів функцій дозволяє зменшити кількість проміжних змінних та працювати з об'єктами чи даними в більш зручному та послідовному способі. Це особливо корисно при роботі з бібліотеками чи інтерфейсами, де послідовні операції можна об'єднувати в ланцюжки для створення більш складних функціональних конструкцій або обробки даних.

Змінні контексту (Context Variables) в нашому дослідженні ми декларуємо як змінні, що визначають контекст або оточення, в якому виконується код внутрішньої DSL. Їхня роль полягає в передачі та збереженні інформації про стан або параметри, які впливають на виконання програми, в рамках обмеженого контексту DSL.

У користувацьких паттернах проектування внутрішньої DSL, контекстні змінні використовуються для передачі та зберігання інформації, специфічної для даних областей, у яких використовується мова. Вони можуть бути використані для управління станом виконання, передачі параметрів, зберігання конфігураційних параметрів або контролю за поточним контекстом виконання коду.

Використання контекстних змінних дозволяє забезпечити зручний інтерфейс для користувачів DSL, де вони можуть взаємодіяти з кодом на більш високому рівні абстракції, використовуючи змінні, які відображають поточний стан або параметри виконання їхніх завдань. Це дозволяє покращити читабельність та зрозумілість коду DSL та сприяє більш ефективному використанню мови у відповідній області застосування.

Вкладені конструктори (nested builders), ми визначаємо як підхід у програмуванні, де конструктори чи методи використовуються для створення складних об'єктів, що містять вкладені частини або компоненти. Цей підхід є особливо корисним у випадках, коли потрібно створити об'єкт зі складною структурою, яка має багато взаємопов'язаних частин. Вкладені конструктори дозволяють послідовно створювати об'єкти, додаючи до них компоненти чи властивості шляхом виклику відповідних методів чи конструкторів.

Це може виглядати як вкладення конструкторів одного об'єкту у методи іншого, що дозволяє створювати глибокі ієрархії об'єктів зрозумілим та лаконічним способом. Використання вкладених конструкторів полегшує процес створення складних об'єктів, забезпечуючи зручний інтерфейс для додавання компонентів чи властивостей до об'єкту. Це особливо корисно в областях, де створення складних структур даних є звичайною практикою, таких як робота з конфігураціями, побудова візуальних елементів інтерфейсу чи створення складних об'єктів даних.

До вбудованих можливостей мови Kotlin, котрі на основі проведеного аналізу ми визначаємо як ті, що дозволяють проектувати внутрішню предметно-орієнтовану мову ми відносимо наступний функціонал мови: лямбда-вирази (зокрема лямбда з приймачем), анотація `@DslMarker`, перевантаження операторів (зокрема оператора виклику функції – “()” (invoke)), користувацькі гетери та сетери, як частина реалізації вищезгаданого патерну контекстних змінних, оператор присвоєння, функціонал перелічень (enums), функції та змінні розширення, інфіксна нотація та її комбінації з можливостями функцій розширення, комбінування інфіксних виразів для формування “речень”.

Таким чином використовуючи вбудовані можливості мови Kotlin для проектування внутрішньої DSL в контексті Android розробки, можна створити функціонал, що інтуїтивно відображає бізнес-логіку.

Наприклад, при розробці інтерфейсу користувача можна використовувати лямбда-вирази з отримувачем для опису конфігурації віджетів, анотацію `@DslMarker` для забезпечення чистоти DSL, а також перевантаження операторів

для спрощення синтаксису. Користувацькі гетери та сетери можуть використовуватися для ефективного управління станами віджетів, тоді як інфіксна нотація у поєднанні з функціями розширення полегшує читабельність коду, дозволяючи формувати логічні "речення", що відображають бізнес-процеси, тощо.

```

@DslMarker
annotation class LayoutDsl // Анотація @DslMarker обмежує область видимості DSL

@LayoutDsl
class LayoutBuilder {
    fun textView(init: TextView.() -> Unit) {
        val textView = TextView()
        textView.init() // Лямбда з отримувачем для конфігурації TextView
        // Додати textView до layout
    }

    // Метод для створення інших UI елементів
}

@LayoutDsl
class TextView {
    private var _text: String = ""

    infix fun withText(text: String) {
        _text = text
        // Інфіксна функція для встановлення тексту
    }

    infix fun withStyle(style: Style) {
        // Інфіксна функція для встановлення стилю
        applyTextAppearance(style)
    }

    private fun applyTextAppearance(style: Style) {
        // Логіка стилізації тексту
    }
}

enum class Style {
    BOLD, ITALIC // Використання enum для визначення стилів
}

// Лямбда з отримувачем для створення layout
fun layout(init: LayoutBuilder.() -> Unit): LayoutBuilder {
    val builder = LayoutBuilder()
    builder.init()
    return builder
}

// Використання DSL
val myLayout = layout { this: LayoutBuilder
    textView { this: TextView
        // Використання інфіксної функції для тексту
        this withText "Welcome to Kotlin DSL"
        // Використання інфіксної функції для стилю
        this withStyle Style.BOLD
    }
}

```

Рисунок 2.3 – Приклад використання можливостей мови Kotlin для створенні внутрішніх DSL

У цьому прикладі використовуються лямбда-вирази з отримувачем для конфігурації TextView, анотація @DslMarker для обмеження області видимості DSL, користувацькі гетери та сетери для TextView, enum для стилізації тексту та інфіксна нотація для більш читабельного коду.

```

class StringDemo {
    infix fun addString(string: String) { /* певна логіка */ }

    fun build() {
        this addString "1234" // коректний вираз
        addString( string: "1234") // коректний вираз
        add "abc" // некоректний вираз: приймач має бути явно визначений
    }
}

```

Рисунок 2.4 – Демонстрація обмеження використання інфіксної нотації в мові програмування Kotlin

В ході написання даного прикладу було виявлено незначний мінус, в контексті використання інфіксних функцій – для їх виклику необхідні дві складові – викликаючий об’єкт та параметр функцій, тобто що інфіксні функції завжди вимагають вказівки і приймача. Так, при виклику методу на поточному приймачі з використанням інфіксної нотації, необхідно використовувати його явно. Це потрібно для забезпечення однозначного розбору компілятором [55].

2.5 Висновки до розділу

В ході роботи над розділом нами було проведено аналіз видів та застосування сучасного інструментарію для розробки предметно-орієнтованих мов. Для створення зовнішніх мов було розглянуто можливості для створення текстових, графічних та проєкційних мов.

Найпопулярнішими інструментами для створення текстових мов є: ANTRL, Xtext та Spoofox. Для потреб Android-розробки на нашу думку найбільш доцільним буде використання ANTLR через його пряму підтримку інтерпретації на сімействі мов, що працюють на базі Java Virtual Machine, прикладом якої є мова програмування Kotlin.

В ході аналізу нами визначено, що графічні мови, як й інструментарій для їх створення надзвичайно складно інтегрувати у процес вирішення потреб Android-платформи, проте зазначимо, що найпопулярнішими інструментами в цій царині є GMF та MetaEdit+ Workbench.

Проєкційні редактори в свою чергу є надзвичайно потужними і водночас складними інструментами через роботу напряму з абстрактним синтаксичним деревом. Нами було розглянуто найбільш популярний інструмент для цих задач – JetBrains MPS, функціонал котрого задовольняє більшість потреб, з котрими стикаються користувачі при розробці власної мови. Враховуючи нашу завдання щодо дослідження адаптації даного інструментарію до потреб Android-розробки варто зазначити, що у випадку досліджуваного редактора проєкційних мов JetBrains MPS завдяки інструменту LionWeb є можливість експорту моделей до

Kotlin, що створює можливості для подальшої інтерпретації власної мови на стороні Android-пристрою, котрий виконує Kotlin-код додатку.

Також нами було розглянуто можливості мови програмування Kotlin для створення внутрішньої предметно-орієнтованої мови. Дана мова підтримує створення визначених нами патернів проектування, що є необхідними для створення внутрішньої DSL, а також має низку вбудованих інструментів що спрощують та заохочують розробників до створення внутрішніх предметно-орієнтованих мов: лямбди з приймачем, інфіксна нотація, анотація `@DslMarker` тощо.

Підсумовуючи даний розділ зазначимо, що у процесі аналізу сучасних інструментальних засобів для розробки DSL ми розглянули низку різноманітних інструментів та зіставили їх особливості порівняльну таблицю 2.1.

Таблиця 2.1

Порівняльний аналіз розглянутого інструментарію

Інструмент	Основне призначення	Підтримка мов програмування	Особливості для DSL	Інтеграція з Android
ANTLR	Парсер-генератор	Java, C#, Kotlin, Python і т.д.	Висока гнучкість	Обмежена
Xtext	Фреймворк для створення DSL на основі парсеру	Java, інтеграція з Eclipse	Висока гнучкість	Критично обмежена
Spoofax	Фреймворк для створення DSL	Різні мови	Багатомовність	Критично обмежена
GMF	Редактор моделей графічних мов	Java	Графічні DSL	Відсутня
MetaEdit+	Графічне моделювання та метамоделювання мов	Спеціалізовані мови	Сильна спеціалізація	Відсутня
Jetbrains MPS	Проекційне моделювання	Java, Kotlin, інші	Висока адаптивність	Можлива
Kotlin DSL	Внутрішні DSL для Kotlin	Kotlin	Інтегрованість	Пряма

Таблиця висвітлює ключові характеристики розглянутого інструментарію та потенціал щодо інтеграції з Android-розробкою. Виходячи з результатів аналізу ANTLR, JetBrains MPS та Kotlin DSL на нашу думку є найбільш вдалим та актуальними інструментами в царині Android-розробки.

3 ЗАСТОСУВАННЯ DSL У ПРОГРАМУВАННІ ДЛЯ ПЛАТФОРМИ ANDROID

У світлі стрімкого розвитку технологій, що здатні реформувати парадигми програмування, особливу увагу слід приділити застосуванню предметно-орієнтованих мов (DSL) у контексті розробки для мобільної платформи Android. Цей розділ, який є третім у структурі нашої магістерської дисертації, покликаний глибоко зануритись у специфіку та витонченість використання DSL на платформі Android, яка відзначається своєю динамічністю та різноманітністю.

Сьогоднішній розвиток мобільних технологій ставить перед науковою спільнотою нові виклики та завдання, вирішення яких потребує нестандартних підходів і інноваційних рішень. Використання DSL у мобільній розробці, особливо для Android, є одним із таких новаторських підходів, що забезпечує значне полегшення процесу розробки, забезпечуючи при цьому гнучкість та специфічність для задоволення потреб конкретного домену чи аплікації.

Цей розділ спрямований на розгортання та аналіз різних аспектів використання DSL для Android, включаючи, але не обмежуючись, технічними характеристиками, методами впровадження, а також аналізом ефективності таких підходів у контексті реального програмного забезпечення. Особлива увага буде приділена порівняльному аналізу з традиційними методами розробки на Android, розглядаючи переваги та можливі обмеження DSL.

В рамках цього розділу ми зосередимо увагу на таких ключових питаннях: технічні особливості DSL, що використовуються для розробки на Android; методи інтеграції DSL у процес розробки мобільних застосунків; оцінка впливу DSL на продуктивність та якість кінцевого продукту; та врешті-решт, можливі стратегії оптимізації застосування DSL у майбутніх проектах.

Аналізуючи застосування DSL, ми будемо оперувати на перетині теоретичних знань і практичного застосування, намагаючись глибоко вникнути в сутність і вплив цих мов на процес розробки на Android. Основною метою

цього розділу є не лише опис та аналіз вже існуючих рішень, але й виявлення потенціалу для інновацій та подальших досліджень у цій галузі.

В підсумку, розділ 3 покликаний відкрити перед читачем широкі горизонти можливостей та перспектив, які DSL можуть принести у сферу розробки мобільних застосунків, зокрема для платформи Android, поєднуючи глибину наукового аналізу з практичними аспектами впровадження.

3.1 Використання внутрішніх DSL

У межах загального розгляду застосування DSL в контексті програмування та проведеного аналізу в минулих розділах ми визначили, що внутрішні DSL є важливим елементом у сфері програмної інженерії, оскільки вони пропонують спосіб розширення та кастомізації існуючих мов програмування для вирішення специфічних завдань, притаманних конкретним доменам чи аплікаціям.

В контексті Android, внутрішні DSL використовуються для створення більш експресивних та зосереджених на домені рішень, виокремлюючи бізнес-логіку від платформно-залежних аспектів. Це може включати автоматизацію складних процесів, оптимізацію робочих потоків, що призводить до підвищення продуктивності розробників та якості кінцевих продуктів.

Основна увага в цьому підрозділі буде приділена аналізу та обговоренню конкретних прикладів внутрішніх DSL, які були успішно інтегровані в процес розробки Android-додатків.

Оскільки компанія Google надає в першу чергу JVM-залежні програмні інтерфейси для створення додатків, котрі можна використовувати здебільшого на мовах Java та Kotlin, а також враховуючи той факт, що з 2019-го року Google рекомендує мову Kotlin для розробки нативних Android-застосунків – даний аналіз буде зосереджуватись лише на мові програмування Kotlin [56], котра враховуючи свої синтаксичні особливості, розглянуті у минулих розділах надає широкий спектр можливостей для створення внутрішньої DSL. Застосування внутрішньої DSL на їх основній мові дозволить розробникам більш інтуїтивно та

ефективно взаємодіяти з компонентами програмними інтерфейсами для розробки додатків на платформі Android.

Цей підрозділ допоможе глибше зрозуміти, як внутрішні DSL можуть спростити та поліпшити процес розробки Android-додатків, а також висвітлити потенціал для інновацій та оптимізації у майбутніх проектах.

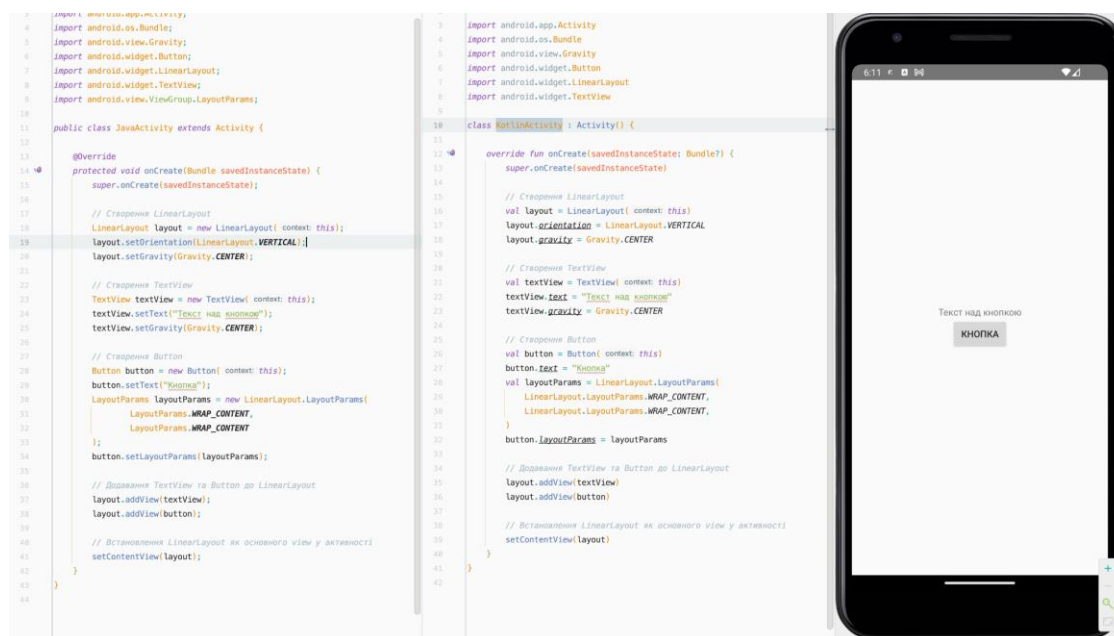
Розробка користувацького інтерфейсу додатку, для будь-якого мобільного застосунку має критичне значення та здатна визначати успіх або невдачу в очах кінцевого користувача.

Процес проектування інтерфейсу є надзвичайно важливим, але часто трудомістким завданням, яке вимагає від розробників не лише технічних знань, але й розуміння принципів дизайну та взаємодії користувача з програмою. Використання спеціалізованих DSL у цьому контексті може значно спростити цей процес, надаючи розробникам потужні інструменти для ефективного та інтуїтивно зрозумілого проектування інтерфейсів.

В цьому підпункті ми розглянемо різні DSL, які були розроблені для вирішення конкретних завдань, пов'язаних із створенням інтерфейсу користувача в мобільних додатках. Ми оцінимо, як ці мови впливають на процес розробки, зокрема, на скорочення часу розробки, підвищення гнучкості в дизайні, а також на покращення взаємодії з користувачем.

Програмні можливості для створення інтерфейсу Android-додатків надані компанією Google доволі широкі. Історично Google надала програмістам систему створення об'єктів на екрані, атомарна одиниця даного об'єкту називалась View, а конструкт, котрий задавав правила, за якими декілька View розміщуються в просторі називався ViewGroup [57].

Створення, та конфігурування View та ViewGroup, відбувалося або безпосередньо з Java (а згодом й Kotlin) коду самого додатку, або комбіновано з допомогою спеціально створеної XML-структурної предметно орієнтованої мови.



Риснок 3.1 — Приклад створення інтерфейсу користувача за допомогою View-системи на мовах Java та Kotlin

В часи розробки Android ОС, створення фреймворку для розробки інтерфейсу в подібному імперативному та об'єктно орієнтованому стилі було стандартною практикою. Подібний синтаксис можна побачити наприклад в популярних десктопних фреймворках, таких як Qt, JavaFX, Windows Forms та інші [58].

З часом стало зрозуміло, що даний підхід створює забагато зайвого та шаблонного коду. Таким чином в світі клієнтської розробки почали набувати популярності декларативні фреймворки для створення користувацького інтерфейсу. Даний тренд здебільшого походив з веб-розробки, де зародилась ціла конкурентна ніша декларативних фреймворків, як от ReactJS [59]. Таким чином, в сучасному контексті розробки Android-додатків з'явилася потреба в нових підходах до програмного створення інтерфейсу користувача, які б відповідали сучасним трендам розробки та вирішували існуючі недоліки імперативного об'єктно-орієнтованого підходу.

Таким чином, на ринку програмних рішень з'явилась незліченна кількість продуктів з відкритим вихідним кодом, головною метою яких було спрощення роботи з наявною View-системою в екосистемі Android. Серед наявних

розширень слід виділити дві, котрі набули найбільшої популярності в середовищі розробників: Anko та Splitties [60, 61].

Розробка першого рішення проводилась переважно під патронажем компанії JetBrains, розробників мови програмування Kotlin, і була покликана продемонструвати можливості цієї мови в створенні внутрішніх предметно орієнтованих мов в домені Android-розробки. Дане рішення набуло надзвичайної популярності. Фактично рішення від JetBrains додало “синтаксичного цукру” поверх наданою компанією Google View-системою створення інтерфейсів. В деякому сенсі, даний програмний продукт слугував демонстрацією технічних можливостей мови програмування Kotlin у контексті створення внутрішніх DSL, котрі були оговорені в минулих розділах.

Проект Splitties першочергово мав на меті об’єднати різного роду корисні утилітні шаблонні конструкти, що використовуються в Android-розробці. З часом даний програмний продукт фактично почав розростатись настільки, що для окремих частин Android-фреймворку необхідно було створення власної внутрішньої DSL. Однією з таких частин була робота з користувацьким інтерфейсом, котра аналогічно Anko створювалась поверх імперативної View-системи, наданої компанією Google, будувала власне декларативне предметно-орієнтоване рішення.

Обидва ці програмні рішення, фактично, за допомогою можливостей мови Kotlin створювали внутрішню предметно-орієнтовану мову програмування, чим додавали додатковий рівень абстракції до існуючої View-системи користувацького інтерфейсу, що значно спрощувало та пришидшувало розробку додатків.

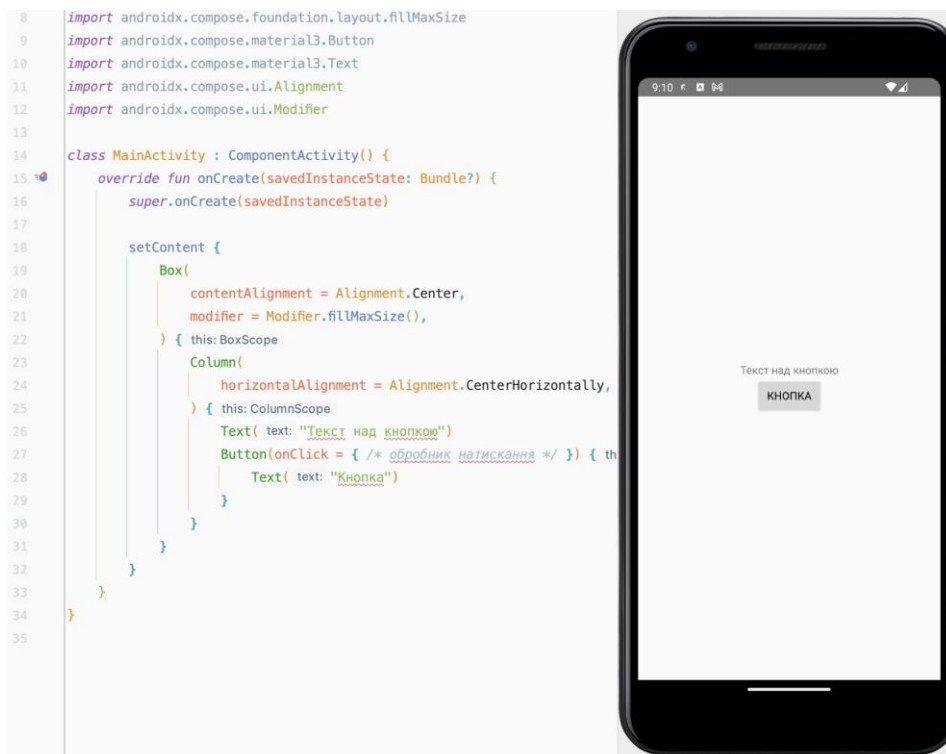


Риснок 3.2 — Приклад створення інтерфейсу користувача за допомогою внутрішньої DSL наданої бібліотеками Anko та Splitties

З часом рішення Anko стало морально застарівати, оскільки компанія Google почала розробку власного декларативного UI-фреймворку. Рішення від JetBrains офіційно стало застарілим (проект помістили в архів), а всі напрацювання наявні в цьому програмному продукті перейшли в бібліотеку Splitties, тому як можна спостерігати на рисунку 3.2 дані рішення в останніх стабільних версіях мають майже ідентичний синтаксис для створення однієї й тієї ж конфігурації екрану.

Фреймворк від компанії Google згодом отримав назву Jetpack Compose та обіцяв кардинально відмінний підхід від View-системи користувацького інтерфейсу. Compose аналогічно до надпопулярного Anko застосовував декларативний функціональний підхід, на протилежність імперативному об'єктно орієнтованому з View-системи, а також на відміну від Anko та Splitties базувався на фундаментально новому підході до створення користувацького інтерфейсу, котрий існує паралельно View-системі та стоїть на тому ж рівні абстракції, надаючи користувачу готові програмні компоненти для створення користувацького інтерфейсу, котрі аналогічно View-системі на нижчому рівні використовують надані операційною системою Android програмні засоби

(Canvas, шейдери OpenGL ES, SKIA, тощо) для продуктивного рендеренгу елементів на екрані користувача.

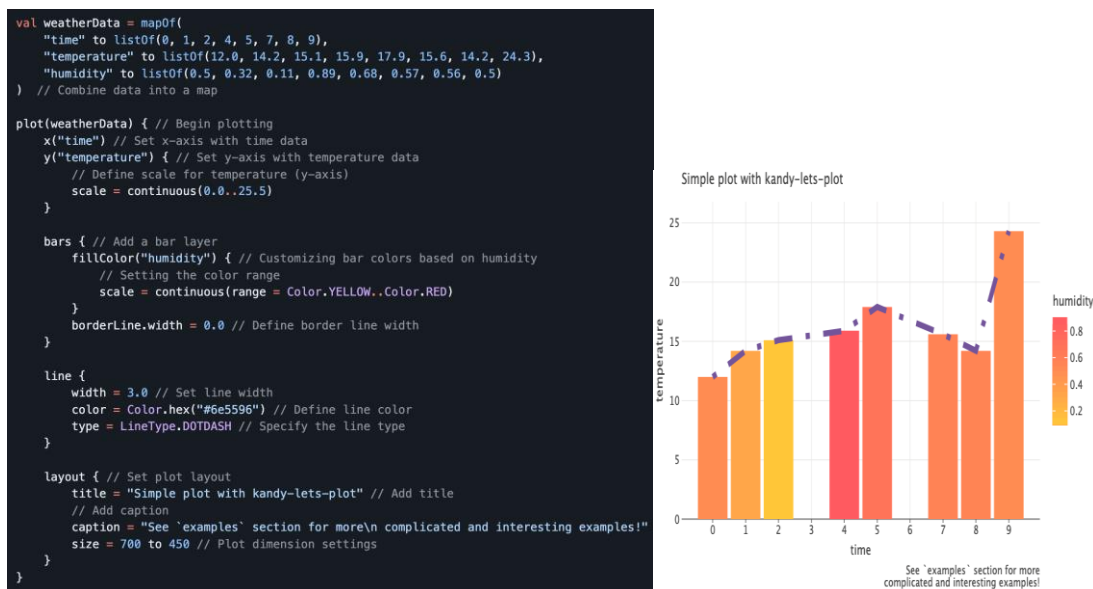


Риснок 3.3 — Приклад створення інтерфейсу користувача за допомогою внутрішньої DSL наданої фреймворком Jetpack Compose

Наразі, Jetpack Compose фактично є індустріальним стандартом, і одним з найяскравіших прикладів застосування внутрішньої DSL в контексті розробки користувацьких додатків для платформи Android, що підтверджується діями компанії Google спрямованими на популяризацію даного продукту. Jetpack Compose, що представляє собою втілення новітніх підходів до створення інтерфейсу користувача, зумовлює переосмислення традиційних методик розробки в контексті мобільних застосунків. Цей фреймворк відкриває нові перспективи за допомогою своєї гнучкості та ефективності наданою не в останню чергу за допомогою створеної внутрішньої предметно-ор, пропонуючи розробникам інтуїтивно зрозумілу та експресивну мову для створення динамічних та реактивних інтерфейсів.

Ще одним прикладом використання внутрішньої предметно-орієнтованої мови є рішення від компанії JetBrains – Kandy [62]. Kandy - це бібліотека

побудови графіків з відкритим вихідним кодом для Kotlin, яка надає потужний і гнучкий DSL для створення графіків і використовує різні популярні рушії [62]. Використання Kandy для побудови графіків є прикладом того, як можна ефективно інтегрувати візуальні елементи з програмною логікою, забезпечуючи при цьому високу продуктивність та гнучкість.



Риснок 3.4 — Демонстративний приклад побудови графіку температури наданий документацією Kandy [62]

В контексті швидкого розвитку технологій та змінних вимог ринку, DSL стають невід'ємною частиною сучасної мобільної розробки, що відкриває нові перспективи для створення інноваційних та ефективних рішень у галузі мобільного програмування, а також пришвидшує впровадження технічних рішень завдяки більш інтуїтивному стилю написання користувацьких програм.

У сфері розробки програмного забезпечення, зокрема у контексті мобільних додатків, тестування відіграє важливу роль у забезпеченні якості та надійності кінцевих продуктів. Цей підпункт присвячений аналізу внутрішніх предметно-орієнтованих мов (DSL) для тестування, які забезпечують значне спрощення процесу валідації функціоналу та виявлення помилок у програмному забезпеченні. Внутрішні DSL для тестування мобільних застосунків демонструють потужний інструментарій, що дозволяє розробникам інтуїтивно і ефективно організовувати та виконувати тестові сценарії.

Застосування DSL у сфері тестування сприяє підвищенню точності та ефективності тестових процедур. Це особливо важливо у контексті мобільних додатків, де вимоги до якості та відповідності очікуванням користувачів є високими. Використання спеціалізованих мов для тестування дозволяє розробникам швидше ідентифікувати та усувати потенційні дефекти, а також забезпечує більш глибоке розуміння поведінки застосунку.

У рамках даного підpunkту, ми розглянемо різні інструменти та мови, розроблені спеціально для потреб тестування. Зокрема, зосередимо увагу на таких аспектах, як можливості автоматизації тестування, інтеграція з різними тестовими фреймворками, здатність до моделювання складних сценаріїв та ефективність виявлення дефектів. Окремо будуть розглянуті можливості внутрішніх DSL у контексті неперервної інтеграції та розгортання, що є ключовими компонентами сучасних методологій розробки програмного забезпечення.

Впровадження внутрішніх DSL для тестування представляє собою стратегічний підхід до підвищення якості мобільних додатків, що сприяє розробці надійних та високопродуктивних застосунків, здатних відповідати високим стандартам та очікуванням кінцевих користувачів.

Під тестуванням, в контексті даної роботи ми розуміємо напряму написання програмного коду, котрий буде перевіряти коректність виконання іншого програмного коду.

Одним з найстаріших інструментальних засобів для тестування коректності створеного користувацького інтерфейсу в екосистемі Android розробки є продукт під назвою Espresso. Espresso – продукт з відкритим вихідним кодом, котрий надає широкий набір можливостей для тестування на реальному пристроїв під управлінням ОС Android, або ж емулятора [63].

Проблема даного рішення є аналогічною до View-системи користувацького інтерфейсу OS Android – даний підхід продукує багато шаблонного повторюваного коду та є морально застарілим, що з одного боку не

розкриває доступні можливості мови програмування Kotlin, з іншого – збільшує інтуїтивний поріг входження нового покоління розробників.

Допоки мова програмування Kotlin не почала займати домінуючі позиції на ринку розробки клієнтських Android-додатків пропозиція внесення змін до існуючої логіки роботи Espresso чи створення додаткового рівня абстракцій було доцільно.

Проте згодом, почали з'являтися різні програмні продукти, основною метою яких, якраз і було спрощення наявної роботи з “UI-тестуванням”. Частина з цих рішень використовуючи механізми, доступні в мові програмування Kotlin створювали внутрішню DSL, використовуючи її як обгортку над Espresso. Деякі приклади цих рішень, ми і розглянемо в даній роботі.

Androidtestktx – доволі просте в застосуванні котра є функцій розширення та інфіксів Kotlin, що зі сторони зовнішнього користувача створює враження спеціалізованої внутрішньої DSL, створеної для покращення читабельності та зменшення шаблонності співставлень та дій Espresso [64].

Дане програмне рішення, як зазначають автори було розроблено для застосування програмного шаблону RobotPattern, оскільки угода про імена функцій передбачає наявність певного семантичного контексту на місці виклику, що вдало реалізується через застосування внутрішньої DSL на мові програмування Kotlin.

```

class MainActivityTest {
    @Test
    fun shouldLoginDemoUser(){
        onView(withId(R.id.activityLoginEditTextUsername)).perform(typeText("dummyUsername"))
        onView(withId(R.id.activityLoginEditTextPassword)).perform(typeText("dummyPassword"))
        onView(withId(R.id.activityLoginBtnLogin)).perform(click())

        Intents.intended(IntentMatchers.hasComponent(MainActivity::class.java.name))
    }
}

class MainActivityTest {
    @Test
    fun shouldLoginDemoUser(){
        typeText("dummyUsername") into text(R.id.activityLoginEditTextUsername)
        typeText("dummyPassword") into text(R.id.activityLoginEditTextPassword)
        click on button(R.id.activityLoginBtnLogin)

        MainActivity::class.verifyThat { itIsDisplayed() }
    }
}

```

Риснок 3.5 — Порівняння синтаксису Espresso та Androidtestktx [64]

Інше популярне рішення – Какао, пропонує кардинально відмінний від Espresso набір API, котрий покликаний збільшити читаємість кодової бази автоматизованого та UI-тестування. Даний фреймворк вимагає визначення окремої додаткової конфігурації екрану, що створює деякі початкові незручності, проте згодом визначення тестових сценаріїв за допомогою декларативного стилю, наданого внутрішньою DSL написаної, аналогічно до Androidtestktx на мові програмування Kotlin, значно спрощує інтуїтивне розуміння коду тестових сценаріїв.

```

@RunWith(AndroidJUnit4::class)
class PriceItemVisibilityTest {
    @get:Rule
    val activityRule = ActivityScenarioRule(SomeActivity::class.java)

    @Test
    fun checkPriceItemIsVisible() {
        onView(
            allOf(
               (withId(R.id.price_item),
                    hasDescendant(withText("Standard Rate")))
            )
        ).check(
            matches(withEffectiveVisibility(Visibility.VISIBLE))
        )
    }
}

class MyActivityScreen : Screen<MyActivityScreen>() {
    val priceItem = KView { withId(R.id.price_item) }
    val standardRateText = KTextView { withText("Standard Rate") }
}

@RunWith(AndroidJUnit4::class)
class PriceItemVisibilityKakaoTest {
    @get:Rule
    val rule = ActivityScenarioRule(SomeActivity::class.java)

    @Test
    fun checkPriceItemIsVisible() {
        MyActivityScreen {
            priceItem {
                isVisible()
                hasDescendant { withText("Standard Rate") }
            }
        }
    }
}

```

Риснок 3.6 — Порівняння синтаксису Espresso та Какао

Варто також зазначити, що дане програмне рішення має окрему підтримку тестів користувачького інтерфейсу, як для View-системи, так і для Jetpack Compose [65]. Рішення для Jetpack Compose визначається окремою бібліотекою, проте має ідейно аналогічний синтаксис в стилі базової бібліотеки.

Програмні рішення, що було розглянуто раніше з одного боку мають доволі вузьку направленість у тестуванні взаємодії користувача з візуальним інтерфейсом, з іншого ж навпаки, завдяки цій можливості можуть тестувати так звані історії користувача, перевіряючи коректність низки послідовних дій користувача, що призводять до зміни стану всієї програмної системи.

Напротивагу такому підходу в тестуванні існує Unit-тестування. Юніт-тестування – процес в програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми. Ідея полягає в тому, щоб писати тести для кожної нетривіальної функції або методу. Це дозволяє досить швидко перевірити, чи не призвело чергову зміну коду до регресії, тобто до появи помилок в місцях програми, які були протестовані, а також полегшує виявлення й усунення таких помилок [66].

Одна з проблем, котру необхідно вирішувати при написанні Unit-тестів – це валідування результату виконання функцій, що стає доволі складною задачею, коли необхідно перевірити, що певна частина коду відпрацювала як очікувалось, а також швидке та зручне створення даних, котрі можуть мати неоднорідне походження. Для вирішення другої проблеми існує два загальновизначених підходи з цих проблем існує два підходи: створення об'єктів заглушок та тестових об'єктів з нереальними даними. Обидва підходи застосовуються на практиці та комбінуються, проте створення системи, що може створювати заглушки для будь-яких об'єктів даних, являє собою нетривіальну задачу.

Програмне рішення з відкритим вихідним кодом MockK має на меті вирішити обидві вищеописані проблеми пов'язані з написанням Unit-тестів, за допомогою внутрішньої DSL на мові програмування Kotlin, що має механізм генерації об'єктів заглушок на основі JVM-рефлексії, а також зручну систему

перевірки та верифікації виконання певних інструкцій в коді, написаному на мові програмування Kotlin [67].

Для демонстрації застосування внутрішньої DSL, автором розроблено тестовий приклад, продемонстрований в додатку, у лівому верхньому куті об'єкт, котрий необхідно протестувати, під ним варіант тесту з використанням MockK DSL, справа – тест без використання сторонніх рішень.

```

class ApiService {
    fun fetchData(): String {
        // Звернення до зовнішнього API
        return "Data from API"
    }
}

class MyService(private val apiService: ApiService) {
    fun processData(): String {
        val data = apiService.fetchData()
        // Обробка отриманих даних
        return "Processed $data"
    }
}

class MyServiceTest {
    @Test
    fun testProcessData() {
        // Створюємо мок для ApiService
        val mockApiService = mockk<ApiService>()

        // Налаштовуємо поведінку моку
        every { mockApiService.fetchData() } returns "Mocked Data"

        // Використовуємо мок у MyService
        val myService = MyService(mockApiService)

        // Перевіряємо результат
        assertEquals("Processed Mocked Data", myService.processData())

        // Перевіряємо, що метод fetchData() був викликаний
        verify { mockApiService.fetchData() }
    }
}

class FakeApiService : ApiService() {
    var fetchDataCalled = false
    var dataToReturn = "Data from API"

    override fun fetchData(): String {
        fetchDataCalled = true
        return dataToReturn
    }
}

class MyServiceTestWithoutMockk {
    @Test
    fun testProcessData() {
        // Створюємо фейкову реалізацію ApiService
        val fakeApiService = FakeApiService()
        fakeApiService.dataToReturn = "Mocked Data"

        // Використовуємо фейковий сервіс у MyService
        val myService = MyService(fakeApiService)

        // Перевіряємо результат
        assertEquals("Processed Mocked Data", myService.processData())

        // Перевіряємо, що метод fetchData() був викликаний
        assertTrue(fakeApiService.fetchDataCalled)
    }
}

```

Риснок 3.7 — Порівняння демонстративних тестів написаних з використанням MockK (зліва) та без використання сторонніх програмних рішень (справа)

Тестовий приклад включає тестування класу MyService, що залежить від зовнішнього API клієнта ApiService. MyService викликає метод fetchData() з ApiService, а потім обробляє отримані дані. В тесті, ми вважаємо необхідним перевірити факт виконання методу fetchData(), та коректність результату виконання методу processData(). Без MockK, тестування MyService вимагає створення ручної заглушка або використання реального екземпляру ApiService. Це може призвести до проблем, якщо ApiService залежить від зовнішніх ресурсів або має складну логіку. Такий підхід може призвести до збільшення часу

виконання тестів та їх залежності від зовнішніх умов. З MockK, ApiService може бути замінений мок-об'єктом.

Завдяки DSL MockK, можливо легко налаштувати поведінку мок-об'єкта (використовуючи every) та перевірити, що певні методи були викликані (verify). Це забезпечує високу гнучкість тестування та ізолює MyService від зовнішніх залежностей, що дозволяє точніше контролювати умови тестування та їх повторюваність, що не в останню чергу досягається за допомогою створеної зручної та інтуїтивно зрозумілої внутрішньої DSL.

Ще однією сферою застосування внутрішніх предметно-орієнтованих мов є **конфігурації збірки** для Android-додатків. В цьому розділі буде розглянуто еволюцію інструментів конфігурації збірки від простих, менш гнучких систем до більш складних та потужних рішень, що дозволяють виконувати широкий спектр завдань, від простого управління залежностями до налаштування складних збіркових сценаріїв.

Особлива увага буде приділена впливу на продуктивність розробки, та можливостям, які вони відкривають для розробників. Таким чином, даний пункт забезпечить глибоке розуміння ключових аспектів конфігурації збірки в контексті Android-розробки, підкреслюючи роль DSL як інструменту для ефективного управління складними проектами в цій області.

На сьогоднішній день інтегроване середовище розробки Android Studio має монопольне становище серед розробників клієнтських застосунків, пропонуючи широкий набір інструментарію для розробки та відлагодження застосунків [68]. Дане програмне рішення за замовчуванням пропонує впровадження системи для збірки проекту Gradle Build Tool.

Gradle Build Tool – починався як проект з відкритим вихідним кодом під керівництвом Ганса Докера (Hans Docker) та Адама Мердока (Adam Murdoch). Початковий успіх проекту Gradle проклав шлях до створення Gradle Enterprise у 2017 році. Як інструмент автоматизації збірки, Gradle підтримує багатомовну розробку програмного забезпечення. Інструмент контролює різні процеси,

пов'язані з розробкою програмного забезпечення. Він автоматизує такі процеси, як компіляція, пакування, тестування, розгортання та публікація [69].

В контексті Gradle, основною перевагою є його гнучкість та можливість адаптації до різних потреб розробників. Gradle використовує внутрішню предметно-орієнтовану мову на базі Groovy або Kotlin, що дозволяє ефективно управляти залежностями та налаштовувати процеси збірки.

Groovy - це потужна, опціонально типізована та динамічна мова зі статичною типізацією та можливостями статичної компіляції для платформи Java, спрямована на підвищення продуктивності розробників завдяки лаконічному, звичному та легкому для вивчення синтаксису.

Вона легко інтегрується з будь-якою програмою на Java і відразу ж надає вашому додатку потужні функції, включаючи можливості написання сценаріїв, створення доменних мов, мета-програмування під час виконання та компіляції, а також функціональне програмування [70].

Можливості надані даними мовами дозволяють розробникам висловлювати складні процедури у зрозумілій та виразній формі. Проте, необхідно зазначити, що висока гнучкість може сприяти зниженню підтримуваності проекту, якщо не дотримуватись певних стандартів або використовувати можливості DSL нераціонально [71].

Особливу увагу у використанні Gradle заслуговує його система плагінів, що дозволяє розширювати можливості DSL. Це включає в себе додавання нових завдань, зміну поведінки існуючих завдань, створення нових об'єктів та ключових слів для опису завдань, які виходять за межі стандартних категорій Gradle.

Такий підхід забезпечує високий рівень абстракції та декларативного опису процесів, що сприяє підвищенню читабельності та підтримуваності кодової бази.

<pre> plugins { java } repositories { mavenCentral() } dependencies { implementation("org.springframework.boot: spring-boot-starter-web:2.3.1.RELEASE") testImplementation("junit:junit:4.12") runtimeOnly("mysql:mysql-connector-java:8.0.20") } </pre>	<pre> apply plugin: 'java' repositories { mavenCentral() } dependencies { implementation 'org.springframework.boot: spring-boot-starter-web:2.3.1.RELEASE' testImplementation 'junit:junit:4.12' runtimeOnly 'mysql:mysql-connector-java:8.0.20' } </pre>
--	---

Риснок 3.8 — Порівняння синтаксису Kotlin DSL та Groovy для налаштування Gradle-конфігурації

Розглянемо, якими саме чином дані мови створюють подібний декларативний синтаксис. Kotlin DSL у Gradle використовує синтаксичні можливості Kotlin, такі як лямбда-вирази з одержувачем, для створення читабельного та виразного DSL. Кожна синтаксична конструкція тут має своє значення.

В Kotlin DSL, “plugins” є функцією, яка приймає лямбда-вираз з одержувачем, де можна визначити плагіни. “java”: Виклик усередині plugins блоку – це функція розширення, що додається DSL для підключення Java плагіна.

repositories { ... }: Цей блок використовується для визначення репозиторіїв, де Gradle може шукати залежності. Як і plugins, repositories є функцією, яка приймає лямбда-вираз з одержувачем. mavenCentral(): Функція, що викликається всередині repositories блоку для додавання Maven Central як репозиторію.

dependencies {...}: Визначає блок для залежностей проекту. Це чергова функція, що приймає лямбда-вираз з одержувачем. implementation(...) – Використовується всередині dependencies блоку для вказування залежностей, необхідних для компіляції та виконання додатку. testImplementation(...), runtimeOnly(...) – інші типи залежностей, які використовуються для тестування та виконання.

Groovy DSL для Gradle використовує динамічні можливості Groovy, такі як клоужери (closures) та динамічне викликання методів, для створення гнучкого DSL. apply plugin: 'java' – Це заява в Groovy DSL для включення певного плагіна. У цьому випадку, це Java плагін. В цій синтаксичній конструкції, фактично

ключове слово “apply” використовується в якості функції, “plugin” – це параметр команди apply, який вказує на те, що ми хочемо застосувати певний плагін., а ‘java’ – це ідентифікатор плагіна, який ми хочемо застосувати. У цьому випадку, це стандартний плагін Gradle для збірки Java-проектів.

repositories { ... }: Так само, як і в Kotlin DSL, цей блок визначає репозиторії. В Groovy, { ... } представляє собою клоужер, який виконується з певним контекстом. dependencies { ... }: Аналогічно Kotlin DSL, визначає залежності проекту. В Groovy DSL, це також клоужер. implementation '...' – в Groovy DSL, строки передаються безпосередньо як аргументи методу. Інші методи виконують аналогічні до прикладу з Kotlin DSL функції.

Зрештою, головна відмінність між Kotlin та Groovy DSL полягає в синтаксичній структурі та механізмах визначення DSL. Kotlin використовує свої функціональні можливості та строгу типізацію, в той час як Groovy покладається на свою динамічність та гнучкість. Обидва підходи надають потужні та виразні засоби для конфігурації збірки Android-додатків.

Таким чином, використання DSL у контексті Gradle Build Tool відкриває нові можливості для розробників Android-додатків, дозволяючи ефективно управляти складними проектами та адаптувати процеси збірки до специфічних вимог та потреб.

Загалом, можемо зробити висновок, що внутрішні DSL, інтегровані безпосередньо в мову програмування, такі як Kotlin DSL, або DSL на базі Groovy використовувани в Gradle, забезпечують гнучкість та виразність, дозволяючи розробникам використовувати знайому синтаксичну структуру та можливості мови спорідненої з JVM-середовищем.

Одним з типових завдань для будь яких розробників є контроль за створенням залежностей між структурними компонентами додатку. Даний процес називають **ін’єкцією залежностей**. Світ розробки мобільних додатків для потреб платформи Android не є виключення з даного правила. Ринок відкритого програмного забезпечення пропонує цілий ряд рішень цієї задачі.

Ми розглянемо два рішення, котрі успішно використовують наявні в Kotlin механізми для створення внутрішньої предметно-орієнтованої мови. Слід також зазначити, що дані продукти набули неабиякої популярності серед спільноти, що є досить незвичною ситуацією для ринку мобільної розробки, оскільки більшість програмних рішень, надає сама ж компанія Google спільно із JetBrains, фактично монополізуючи ринок програмних рішень.

Першим рішенням є Koin – фреймворк, для допомоги в створенні будь-якого типу додатків на Kotlin [72]. В рамках даного дослідження нас цікавить саме можливість використання для розробки мобільних додатків для Android. Також слід зазначити, що Koin розробляється компанією Kotzilla та учасниками з відкритим вихідним кодом [72].

```
val myModule = module {
    single<Repository> { MyRepository() }
    factory { MyViewModel(get()) }
}
```

Риснок 3.9 — Приклад використання Koin для створення об'єкта одинака MyRepository та фабрики об'єктів MyViewModel

У Koin, DSL створюється за допомогою лямбда-виразів з одержувачем, де module функція приймає лямбда-вираз, що діє на контексті модуля. Це дозволяє використовувати single та factory безпосередньо всередині блоку module. Ці ключові слова single та factory є методами розширення, що додаються до контексту модуля, дозволяючи визначати залежності з використанням функціонального синтаксису Kotlin.

Іншим популярним рішенням є Kodein. Дане програмний продукт являє собою дуже простий і водночас дуже корисний контейнер для пошуку залежностей, розробники зазначають, що основною його перевагою є легке використання та налаштування, що не в останню чергу досягається завдяки використанню Kotlin DSL [73]. Розглянемо, як даний фреймворк вирішує аналогічну до прикладу з Koin задачу:

```

val myModule = DI.Module("MyModule") {
    bind<Repository>() with singleton { MyRepository() }
    bind<MyViewModel>() with provider { MyViewModel(instance()) }
}

```

Риснок 3.10 — Приклад використання Kodein для створення об'єкта одинака MyRepository та фабрики об'єктів MyViewModel

Kodein використовує подібний підхід з використанням лямбда-виразів з одержувачем. `DI.Module` є конструктором модуля, що приймає назву та лямбда-вираз для конфігурації залежностей. Методи `bind`, `singleton`, і `provider` є фундаментальними складовими DSL Kodein, що дозволяють виразно визначати, як повинні створюватися та управлятися залежності. В свою чергу функція “with” у Kodein DSL є методом, який входить в ланцюжок викликів DSL для створення визначень залежностей. Синтаксис “with” у Kodein досягається через використання функцій вищого порядку та лямбда-виразів з одержувачем у Kotlin. Це дозволяє створити плавний та інтуїтивно зрозумілий інтерфейс для визначення залежностей. Функція `with` є частиною DSL, що дозволяє розробникам легко та виразно описувати, як об'єкти мають бути створені та управлятися в межах контейнера залежностей Kodein.

Останнім прикладом популярної внутрішньої предметно-орієнтованої мови, що використовується в розробці застосунків під операційну систему Android є фреймворк для створення асинхронних серверних та клієнтських інтернет додатків – Ktor [74].

Ktor, аналогічно до більшості розглянутих в даній роботі програмних продуктів має відкритий програмний код. Також варто зазначити, що дане рішення створюється та підтримується переважно компанією-розробницею мови Kotlin – JetBrains [74]. Розглянемо невеликий приклад використання цього фреймворку.

```

fun main() {
    embeddedServer(Netty, port = 8080) {
        routing {
            get("/") {
                call.respondText("Hello, Android App")
            }
            get("/data") {
                // Припустимо, тут здійснюється вибірка даних для додатку
                val data = fetchDataFromDatabase()
                call.respond(data)
            }
        }
    }.start(wait = true)
}

fun fetchDataFromDatabase(): String {
    // Логіка вибірки даних
    return "Some data"
}

```

Риснок 3.11 — Приклад використання Ktor

У цьому прикладі: `embeddedServer(Netty, port = 8080){...}`: Створює сервер на базі Netty, що слухає на порту 8080. Цей виклик демонструє використання функції вищого порядку з лямбда-виразом. `routing{...}`: Визначає блок для маршрутизації запитів. Це ключова частина Ktor DSL, яка дозволяє декларативно описати маршрути HTTP. `get("/") {...}`: Декларує обробник GET запитів на кореневий шлях ("/"). У середині блоку використовується `call.respondText`, щоб відправити відповідь. `get("/data") {...}`: Додатковий маршрут для обробки GET запитів на шлях "/data". Тут можна реалізувати вибірку даних і відправити їх як відповідь.

Ktor використовує DSL, створений за допомогою функцій розширення і лямбда-виразів з одержувачем у Kotlin. Це дозволяє створювати плавний та зрозумілий інтерфейс для конфігурації сервера та маршрутизації. Лямбда-вирази з одержувачем надають контекст (наприклад, `Application` у `embeddedServer` або `Routing` у `routing`), в якому виконуються відповідні функції та методи DSL. Цей підхід дозволяє розробникам Android-додатків або backend-систем легко створювати та управляти веб-сервісами, використовуючи знайомий синтаксис Kotlin і виразний стиль програмування.

3.2 Використання зовнішніх DSL

Цей підрозділ зосереджений на аналізі використання та імплементації різноманітних зовнішніх DSL, які застосовуються у контексті Android-розробки. Зовнішні DSL надають спеціалізовані та високофункціональні інструменти для конкретних аспектів програмування, що включають, але не обмежуються, створенням користувацьких інтерфейсів, налаштуванням збірки проектів, управлінням базами даних, конфігурацією серверних взаємодій та багатьма іншими.

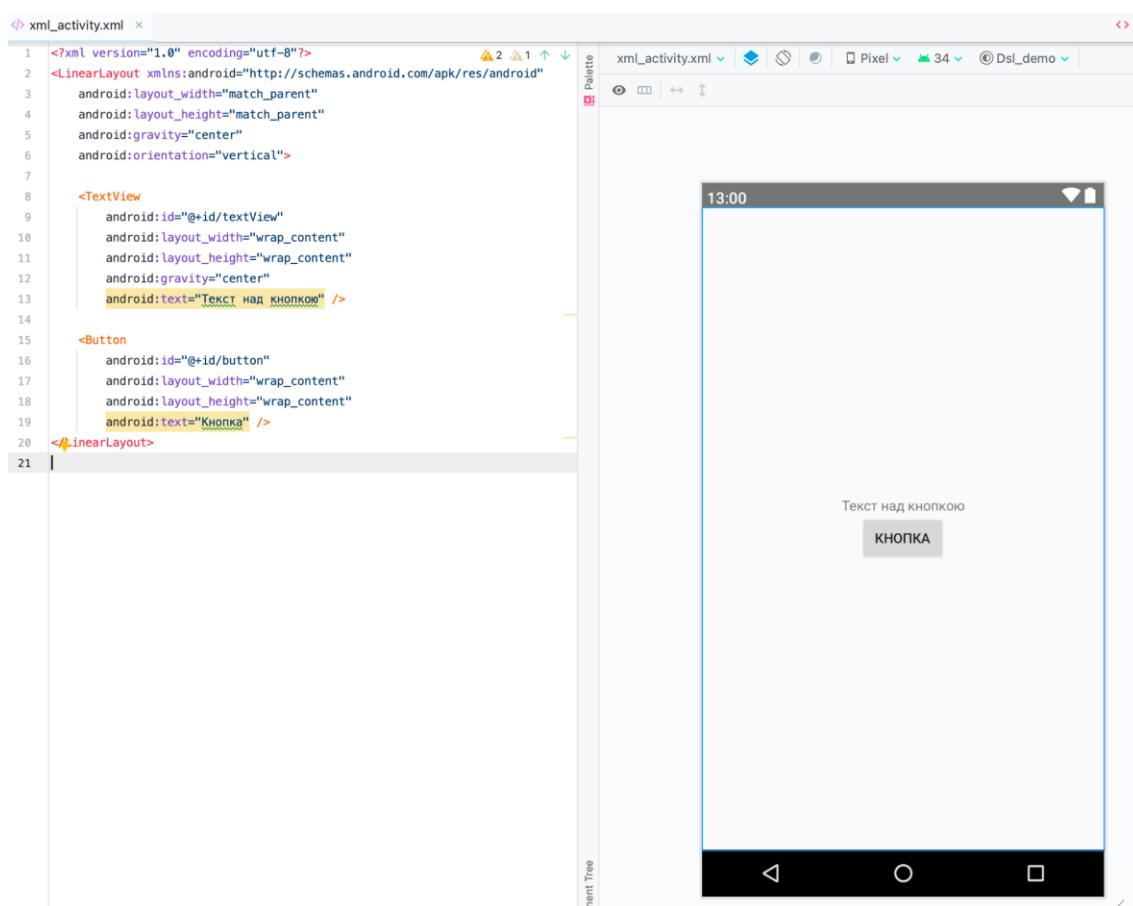
В межах даного підрозділу нами буде розглянуто використання популярних у Android-спільноті зовнішніх предметно-орієнтованих мов.

Один з найпопулярніших напрямків для створення будь-якої предметно-орієнтованої мови є конфігурація користувацького інтерфейсу. В минулих розділах нами вже було розглянуто ряд внутрішніх DSL що вирішують цю задачу в рамках екосистеми Android-розробки. В цьому розділі ми зробимо більш детальний акцент на застосування зовнішньої предметно-орієнтованої мови заснованої на XML, котру початково обрала компанія Google у View-системі для конфігурації користувацького інтерфейсу при розробці Android-застосунків.

Перш ніж проаналізувати синтаксис та застосування даного рішення варто зазначити що слід чітко розділяти конфігураційні файли, та предметно орієнтовані мови побудовані на основі синтаксису наданого стандартами конфігураційних форматів. Так на думку вже відомого нам Фоулера багато конфігураційних файлів мають мовну природу, а отже є DSL. Якщо вони створені у XML їх можна вважати зовнішніми DSL. XML не є мовою програмування; це синтаксична структура без семантики. Тому XML-файли при обробці, зчитує код в токени, а не інтерпретує його для виконання. Обробка в даному контексті по суті є побудовою синтаксичного дерева, тому Фоулер вважає XML синтаксисом-носієм для DSL, подібно до того, як внутрішня мова DSL забезпечена синтаксисом-носієм. Внутрішня DSL також забезпечує семантику носія [15]. Тож зрештую, в контексті даного дослідження ми

вважаємо застосування XML View-системою побудови інтерфейсу користувача окремою предметно-орієнтованою мовою, домен якої полягає в описі поведінки користувацького інтерфейсу при програмуванні застосунків на базі ОС Android.

View-система в Android використовує XML для декларативного визначення користувацького інтерфейсу. XML дозволяє описати макет, стилі, відступи, а також інші властивості елементів інтерфейсу [75]. Розглянемо синтаксис на прикладі наведеного XML-файлу:



Риснок 3.12 — Приклад створення інтерфейсу користувача за допомогою XML в межах View-системи

Для прикладу ми створили аналогічний інтерфейс до того, що вже був створювався за допомогою внутрішніх предметно-орієнтованих мов у минулих розділах, що складається з колонки по центру екрану, що містить текст та кнопку. Для створення еквівалентного інтерфейсу користувача в Android за допомогою XML необхідно буде визначати макет, що буде містити певний набір тегів, що буде відповідати необхідним View та ViewGroup, у нашому випадку це

буде `LinearLayout` та – `TextView` і `Button`. Кожен тег у цьому XML відповідає `View` або `ViewGroup` в Android. Розглянемо тег `<TextView>` – це початковий тег, який вказує на створення нового екземпляру `TextView`. `TextView` є класом у `View`-системі Android SDK, що надає можливість відображення тексту на екрані у мобільному додатку. Тег `<TextView>` в XML-макеті відповідає за створення та конфігурацію цього текстового віджета.

Всередині тегу `<TextView>`, можна визначити різноманітні атрибути для кастомізації зовнішнього вигляду та поведінки текстового віджета. Наприклад, `android:layout_width` та `android:layout_height` визначають розміри `TextView`, `android:text` встановлює текст, який буде відображений, а `android:textSize` визначає розмір шрифту.

Розглянемо більш детально синтаксис атрибутів тегу `TextView`:

"android" – префікс простору імен, який використовується для ідентифікації атрибутів, що належать до Android SDK. У XML, простори імен використовуються для уникнення конфліктів імен між елементами та атрибутами, які могли б мати однакові імена, але різне значення або використання. В Android XML макетах, "android" вказує, що атрибут є частиною стандартної бібліотеки Android.

":" – двокрапка в XML використовується для розділення префікса простору імен від імені атрибуту. Таким чином, коли ми бачимо "android:layout_width" – означає, що атрибут "layout_width" належить до простору імен, який позначений префіксом "android".

"layout_width": Це атрибут, використовуваний в XML макетах Android для визначення ширини `View` або `ViewGroup`. Він вказує на те, як широка повинна бути компонента на екрані. Можливі значення для "layout_width" включають "wrap_content" (ширина, необхідна для вміщення вмісту компоненти), "match_parent" (ширина, рівна ширині батьківської компоненти), або конкретне числове значення в пікселях.

Загалом, можемо зробити висновок, що елементів, що керуються тими ж правилами в XML макетах Android дозволяє точно і гнучко визначати

властивості різних компонентів інтерфейсу користувача, забезпечуючи зручність та інтуїтивність у процесі дизайну інтерфейсів. А також виступає як важливий інструмент зовнішньої предметно-орієнтованої мови, що забезпечує ефективний та інтуїтивно зрозумілий механізм для декларативного опису інтерфейсу користувача та інших аспектів додатку. Цей підхід відіграє ключову роль у розробці мобільних додатків на Android, дозволяючи розробникам чітко відділяти опис інтерфейсу від бізнес-логіки та коду додатку. Особливо значущим є той факт, що XML дозволяє здійснювати складне позиціонування та стилізацію UI компонентів без необхідності втручання у код Java або Kotlin. Це не тільки сприяє більшій чистоті та організованості кодової бази, але й забезпечує високу адаптивність інтерфейсу до різних розмірів екранів та орієнтацій пристрою. Застосування XML як зовнішньої DSL у розробці для Android демонструє його важливість як потужного інструменту, що спрощує та оптимізує процес створення користувацьких інтерфейсів, при цьому забезпечуючи високу гнучкість та можливість кастомізації. Ця мова стала фундаментальною частиною екосистеми Android, відіграючи важливу роль у розвитку сучасних мобільних додатків.

Іншою популярною в контексті програмування для системи Android зовнішньою предметно-орієнтованою мовою є мова запитів до бази даних **SQL**.

SQL – це інструмент для організації, управління та пошуку даних, що зберігаються в комп'ютерній базі даних. Назва "SQL" є аббревіатурою від Structured Query Language (мова структурованих запитів). Як випливає з назви, SQL - це комп'ютерна мова, яку ви використовуєте для взаємодії з базою даних. Насправді SQL працює з одним конкретним типом баз даних, який називається реляційною базою даних. Мова SQL і засновані на ній системи реляційних баз даних є однією з найважливіших базових технологій в комп'ютерній індустрії на сьогоднішній день. За останнє десятиліття популярність SQL стрімко зросла, і сьогодні вона є стандартною мовою комп'ютерних баз даних. ОС Android не є виключенням з цього тренду та широко застосовує мову запитів SQL. В якості реляційної бази даних ОС Android використовує SQLite [76]. SQLite – це

бібліотека на мові C, яка реалізує невеликий, швидкий, автономний, високонадійний, повнофункціональний рушій баз даних SQL. SQLite - це найпоширеніша система управління базами даних у світі. SQLite вбудовано в усі мобільні телефони та більшість комп'ютерів, а також входить до складу незліченної кількості інших програм, якими люди користуються щодня [77]. В межах нашого дослідження ми визначаємо мову структурованих запитів SQL як зовнішню предметно-орієнтовану мову, доменом якої є взаємодія з реляційними базами даних, що використовується в Android розробці за допомогою бібліотеки SQLite.

Ще однією широко-застосовуваною зовнішньою предметно-орієнтованою мовою в контексті програмування Android застосунків є **Shader Language** для графіки, однією з найпопулярніших мов в даній категорії є GLSL – OpenGL Shading Language.

Відповідно до останньої специфікації програма написана мовою GLSL – це набір шейдерів, скомпільованих та зв'язаних між собою, шейдери в свою чергу – це незалежні одиниці компіляції, написані цією мовою [78].

Зазвичай контекст використання спеціалізованих мов шейдерів зводиться до додатків для редагування фото, або ж ігрових застосунків. В обох випадках за допомогою мови шейдерів пишеться цілий фреймворк під специфічні задачі – графічний рушій, проте в контексті даної роботи ми будемо сприймати застосування GLSL у виключно прикладних варіант для високопродуктивного графічного обчислення під час роботи звичайного застосунку. GLSL використовується для написання фрагментних (піксельних) та вертексних (точкових) шейдерів, які визначають, як пікселі та вертекси обробляються та відображаються на екрані. За допомогою шейдерів можна створювати складні візуальні ефекти, такі як світлотінь, текстуровані поверхні, переходи, відблиски та інші динамічні візуальні елементи. Шейдери виконуються безпосередньо на графічному процесорі (GPU), що забезпечує високу продуктивність та ефективність при візуалізації графіки мовою [78]. Одним з прикладів такого застосування є так званий Blur-ефект, або ж ефект замилення [79].



Риснок 3.13 — Приклад використання GLSL коду інтегрованого в Kotlin код в Android застосунку [80]

На прикладі вище ми бачимо яким чином можна використовувати предметно орієнтовану мову GLSL в контексті Android розробки. Технічно ми зазначаємо GLSL-код у змінну типу String мови Kotlin, та згодом за допомогою інтегрованої в OS Android графічної бібліотеки Skia буде компільований та виконаний під час виконання відповідного Kotlin коду. Варто також зазначити що Skia - це бібліотека 2D-графіки з відкритим вихідним кодом, яка надає загальні API, що працюють на різних апаратних і програмних платформах. Вона слугує графічним рушієм для Google Chrome та ChromeOS, Android, Flutter та багатьох інших продуктів. Skia спонсорується і управляється компанією Google, але доступна для використання будь-кому за ліцензією BSD Free Software License. Згідно філософії відкритого програмного забезпечення, хоча розробкою основних компонентів займається команда розробників Skia, будь-хто може запропонувати зміни до даного продукту [81].

Подібний підхід з використанням шейдерів та графічної бібліотеки Skia у якості виконуваного середовища використовує в деяких місцях й вже відомий нам за минулими розділами фреймворк Jetpack Compose. Наприклад, починаючи з версії 13 ОС Android доступний модифікатор незалежних блоків Compose-

елементів під назвою Haze, котрий аналогічно використовує предметно-орієнтовану мову GLSL для рендерингу ефекту замилення [82].

```
vec4 main(vec2 coord) {
    vec4 c = content.eval(coord);

    if (!rectContains(rectangle, coord)) {
        // If we're not drawing in the rectangle, return transparent
        return vec4(0.0, 0.0, 0.0, 0.0);
    }

    vec4 b = blur.eval(coord);

    // Add noise for extra texture
    float noiseLuminance = dot(noise.eval(coord).rgb, vec3(0.2126, 0.7152, 0.0722));
    // We apply the noise, with the given noiseFactor
    float n = min(1.0, noiseLuminance) * noiseFactor;

    // Apply the noise, and shift towards `color` by `colorShift`
    return b + n + ((color - b) * colorShift);
}
```

Риснок 3.14 — Приклад GLSL коду з бібліотеки Jetpack Compose для створення Blur-ефекту [82]

Підсумовуючи, можемо зазначити, що найбільш популярною мовою шейдерів, що використовується при програмування під платформу Android є GLSL, котра є предметно-орієнтованою мовою в домені високопродуктивних графічних обчислень.

Варто зазначити, що існує також низка закритих специфічних зовнішніх DSL, розроблених окремими компаніями для Android-розробки, часто створюються для вирішення унікальних задач або потреб, які виходять за рамки стандартних мов програмування та інструментів. Однак, інформація про такі DSL може бути обмеженою через комерційну таємницю або специфіку внутрішньої імплементації. Незважаючи на це, нами було проаналізовано загальні приклади, де компанії можуть розробляти власні DSL.

Одним з популярних варіантів закритих предметно-орієнтованих мов є DSL для серверного визначення інтерфейсу: У компаній, які розробляють складні або високо кастомізовані інтерфейси, може бути свій DSL для опису UI компонентів, анімацій та взаємодій, котрий конфігурується прямо сервером та пишеться непрограмістами, а прямо замовниками бізнес вимог. Ще одним прикладом є DSL для автоматизації тестування. Багато компаній розробляють власні DSL для опису та автоматизації тестових сценаріїв. Наприклад, DSL, який

дозволяє описувати користувацькі взаємодії з UI у вигляді високорівневих команд, спрощує процес написання тестів. Для складних додатків, які вимагають гнучкої конфігурації, компанії можуть створювати DSL для опису налаштувань. Це може включати конфігурацію мережевих запитів, обробку даних або взаємодію з певними компонентами системи. У додатках, що містять складну бізнес-логіку, може бути створений DSL для декларативного визначення правил і процесів. Такий підхід дозволяє бізнес-аналітикам та іншим не-технічним учасникам проекту вносити зміни без необхідності зміни коду програми.

Важливо відзначити, що більшість з цих DSL є внутрішніми інструментами компаній і рідко публікуються або використовуються поза межами цих організацій. Вони створюються з метою підвищення продуктивності, якості та гнучкості розробки відповідних продуктів.

3.3 Висновки до розділу

Підсумовуючі, нами було створено порівняльні таблиці 3.1 та 3.2 оглянутих програмних рішень для розглянутих зовнішніх та внутрішніх предметно-орієнтованих мов.

Для внутрішніх мов порівняння охоплює короткий опис сфери застосування та вирішуваних проблем наданих програмних рішень, а також опис синтаксичних особливостей мови програмування Kotlin, що використовувались для створення внутрішньої предметно-орієнтованої мови. Ми розглянули переваги використання внутрішніх DSL на базі Kotlin для різних аспектів розробки Android-додатків. Застосування Kotlin DSL дозволяє значно спростити та оптимізувати процес розробки, зокрема завдяки зменшенню кількості шаблонного коду та підвищенню читабельності.

Більша частина з цих рішень використовують вбудовані можливості мови програмування Kotlin такі як функції розширення та лямбди з розширенням, проте один з розглянутих фреймворків – Jetpack Compose додатково використовує плагін для компілятора мови Kotlin, що працює під час роботи програми та генерує Java байт-код.

Порівняння популярних внутрішніх DSL в розробці Android-застосунків

Програмний продукт	Сфера застосування (домен)	Вирішувана проблема	Використані синтаксичні особливості Kotlin
Anko	Дизайн UI для Android	Спрощує розробку UI, зменшує шаблонний код	Розширення Kotlin, лямбда-вирази, безпечні конструктори типів
Splitties	Розробка Android	Набір практичних бібліотек Kotlin Android	Функції розширення Kotlin, лямбда-вирази для інтуїтивного синтаксису
Jetpack Compose	Дизайн UI для Android	Сучасний інструментарій UI для Android	Декларативний синтаксис Kotlin, композиційні функції, що вбудовані у мову Kotlin завдяки плагінам компілятора
Kaako	Тестування UI	Поліпшення фреймворку тестування Espresso	Використання особливостей DSL Kotlin для більш зрозумілих тестів
Androidtestctx	Тестування UI	Спрощує і покращує тестування з Espresso	Функції розширення Kotlin, інфіксна нотація для DSL
Mockk	Модульне тестування	Інструменти для створення заглушок (mock) у Kotlin	Вбудовані функції Kotlin, лямбда-вирази, анотації
Kotlin DSL для Gradle	Конфігурація збірки	Спрощує скрипти збірки Gradle з синтаксисом Kotlin	Безпечні конструктори типів Kotlin, лямбда-вирази для чітких скриптів
Groovy DSL для Gradle	Конфігурація збірки	Динамічна мова для конфігурації збірок Gradle	Динамічні особливості Groovy, акцент на читабельності та гнучкості
Koin	Ін'єкція залежностей	Не перевантажене рішення для ін'єкції залежностей для Kotlin	DSL з використанням лямбда з приймачем і функцій розширення
Kodein	Ін'єкція залежностей	Прагматичне рішення для ін'єкції залежностей	Безпечні конструктори типів, делеговані властивості у DSL
Ktor	Мережева конфігурація	Рішення для створення асинхронних серверів і клієнтів	Функції розширення для створення стислого DSL

Загалом такі фреймворки, як Kodein, Jetpack Compose, та інші, ефективно використовують синтаксичні особливості Kotlin, створюючи більш інтуїтивно зрозумілий та виразний DSL. Це не лише сприяє поліпшенню продуктивності розробників, але й відкриває нові можливості для створення більш гнучких та масштабованих додатків. Використання DSL у Kotlin являє собою важливий крок у напрямку модернізації підходів до розробки програмного забезпечення.

Для розглянутих зовнішніх предметно-орієнтованих мов: XML, SQL, GLSL та закритих зовнішніх DSL ми маємо зіставлену порівняльну таблицю. Ми

дослідили, як ці мови спеціалізуються на вирішенні певних задач і сприяють підвищенню ефективності та гнучкості в процесах розробки.

Таблиця 3.2

Порівняння популярних зовнішніх DSL в розробці Android-застосунків

DSL	Область Застосування	Вирішувані Проблеми	Опис
XML	Дані та конфігурації	Стандартизація обміну даними	Мова розмітки, яка використовується для зберігання та передачі даних
SQL	Бази даних	Маніпуляція та управління даними	Мова запитів для роботи з реляційними базами даних
GLSL	Графіка	Розробка шейдерів	Мова програмування шейдерів для написання коду в контексті 3D-графіки
Зовнішні закриті DSL	Специфічні домени	Вирішення унікальних завдань	Спеціалізовані мови, розроблені для конкретних завдань або галузей

Було проведено глибокий аналіз зовнішніх предметно-орієнтованих мов, таких як XML, SQL, GLSL, та специфічних закритих DSL, у контексті їх використання в розробці для Android. Особливу увагу було приділено аналізу специфічних характеристик кожної мови, включаючи їхні переваги та обмеження. Цей аналіз не лише підкреслює важливість зовнішніх DSL у розробці програмного забезпечення, але й вносить значний вклад у розуміння того, як ці мови можуть бути ефективно інтегровані в різні аспекти розробки. Що відображає ключові аспекти використання зовнішніх DSL у контексті сучасного програмування для потреб екосистеми Android, що є невід'ємною частиною загальних цілей і задач нашої роботи.

4 РЕАЛІЗАЦІЯ ПРОГРАМНИХ ПРОТОТИПІВ

Під час розробки мобільних застосунків розробниками у зв'язці із замовниками проектуються вимоги до різноманітних частин додатку. Типовим завданням, котре з'являється перед командою створення додатку, незалежно від його домену, стає створення та проектування екрану або навіть групи екранів, що мають повторно використовувані елементи, на кшталт різноманітних інформаційних покажчиків у вигляді карток.

Даний функціонал може бути розташований, до прикладу, на головному екрані, агрегуючи інформацію з різних джерел додатку, та, в залежності від визначених бізнесом вимог, мати певну поведінку: перелік станів, умови їх зміну, порядок, тощо.

Для більшої гнучкості в тестуванні, або впровадженні нових карток сторона замовників може висувати вимоги у вигляді доступної переконфігурації цих карток без оновлення додатку. Дану проблему в уявній клієнт-серверній архітектурі можна вирішити за допомогою опису конфігурацій у різноманітних форматах як-от: JSON, XML, YAML тощо, котрі можуть редагуватись з конфігураційної панелі, що будучи пов'язаною з сервером надсилає конфігурацію на клієнти.

Додатково при такій реалізації необхідно обов'язково створити опис очікуваної поведінки у вимогах до задачі. Проблемою такого підходу є відсутність набору правил, що ідентифікується як однозначний контракт між замовником, клієнтом та сервером, що може стати проблемою в умовах сучасних гнучких стандартів створення програмного забезпечення, адже такий формат не дає жодної гарантії, що після чергових змін до програмного продукту на стороні клієнта чи сервера функціонал буде працювати саме так, як це закладалась при проектуванні. Прикладом проблеми, що може статись при такому підході є некоректний літерал у JSON-файлі, котрий, до прикладу не буде визначений бекендом як помилковий при перевірці валідності JSON-файлу, проте може

призвести до непередбачуваної поведінки на стороні клієнта від збоїв під час виконання до аварійного завершення роботи додатку.

Для вирішення цієї проблеми, а також демонстрації можливостей, що надаються предметно-орієнтованими мовами нами було створено програмні прототипи предметно-орієнтованих мов для конфігурацій карток. Завдяки обмеженням, котрі ми власноруч можемо запровадити за допомогою специфічної граматики та синтаксису визначеної нами мови, ми усуваємо можливість неоднозначностей трактування конфігураційних файлів, та як наслідок створюємо однозначний контракт в кодї, а не документації щодо того, яким чином спроектоване рішення.

У демонстративному порядку, нами було визначено ряд вимог до програмного прототипу:

1. Мета мови – створення мови для управління станами карток у мобільному додатку Android, до прикладу, для домену фітнесу та тренувань.
2. Стани карток – можливість визначати різні стани для кожної картки, наприклад: активний, неактивний, завершений.
3. Переходи між станами – синтаксис для опису переходів між станами на основі подій.
4. Залежності між картками – визначення залежностей карток одна від одної на основі їх станів.
5. Система пріоритетів – механізм визначення пріоритетів карток.
6. Умовні переходи – Синтаксис для опису умовних переходів між станами залежно від подій та умов.

Для програмної реалізації зовнішньої предметно-орієнтованої мови нами було обрано інструмент опису граматики та генерації синтаксичного аналізатора – ANTLR. Внутрішню ж предметно-орієнтовану мову ми будемо створювати за допомогою механізмів мови Kotlin.

4.1 Програмний прототип зовнішньої DSL

Відповідно до даного завдання нами була спроектована, граматики CardStateDslGrammar, написана для ANTLR, котра служить формальним визначенням мови програмування, яка специфічно призначена для опису станів та переходів у контексті "карток".

```

1  grammar CardStateDslGrammar;
2
3  dsl: cardDeclaration+;
4
5  cardDeclaration: 'Card' ID 'Priority' INT '{' stateDeclaration+ '}';
6
7  stateDeclaration: 'State' ID '{' transitionDeclaration+ '}';
8
9  transitionDeclaration: 'on' ID 'become' ID ';'
10                       | 'if' '(' expression ')' 'become' ID ';';
11
12  expression: ID 'is' ID;
13
14  ID: [a-zA-Z]+;
15  INT: [0-9]+;
16
17  WS: [ \t\r\n]+ -> skip;

```

Рисунок 4.1 — Опис граматики за допомогою ANTLR

Розглянемо кожен компонент цієї граматики детально:

- `grammar CardStateDslGrammar;` – конструкція визначає назву граматики - CardStateDslGrammar. У контексті ANTLR, `grammar` є ключовим словом, що ініціює оголошення граматики, а `CardStateDslGrammar` є унікальною назвою, що ідентифікує дану граматику.
- `dsl: cardDeclaration+;` – в даній конструкції `dsl` є вхідним правилом, яке визначає основну структуру DSL. Символ “+” означає, що у DSL повинна бути присутня хоча б одна декларація картки (`cardDeclaration`).
- `cardDeclaration: 'Card' ID 'Priority' INT '{' stateDeclaration+ '}';` – правило, що визначає структуру декларації картки. Ключові слова `Card` і `Priority` вказують на початок визначення картки та її пріоритет відповідно. `ID` та `INT` є плейсхолдерами для ідентифікатора картки та числового значення

пріоритету. Символи “{“ та “}” використовуються для обмеження блоку станів картки.

- `stateDeclaration: 'State' ID '{' transitionDeclaration+ '}'`; – визначає структуру декларації стану в рамках картки. `State` є ключовим словом, що позначає початок визначення стану, за яким слідує ідентифікатор стану (ID).
- `transitionDeclaration: 'on' ID 'become' ID ';' | 'if' '(' expression ')' 'become' ID ';'` – правило для визначення переходів між станами. Воно дозволяє визначати перехід на основі події (`on ID become ID`) або умови (`if '(' expression ')' become ID`).
- `expression: ID 'is' ID`; – визначає логічний вираз для умовних переходів, де `ID 'is' ID` формує простий умовний вираз.
- `ID: [a-zA-Z]+`; – являє собою лексичне правило, що визначає ідентифікатори (ID). Ідентифікатори складаються з однієї або більше літер.
- `INT: [0-9]+`; – лексичне правило для визначення цілих чисел (INT), які складаються з однієї або більше цифр.
- `WS: [\t\r\n]+ -> skip`; – правило дозволяє ігнорувати пробіли, табуляції, символи нового рядка та повернення каретки. Конструкція “`-> skip`” вказує на те, що відповідні символи не будуть включені в кінцевий токенований вивід.

Ця граMATика представляє собою формалізований опис структури мови, що призначена для визначення поведінки карток із різними станами та переходами між ними. Вона ілюструє використання контекстно-вільної граMATики для створення виразної та гнучкої мови DSL, що може бути застосована в різних областях програмування.

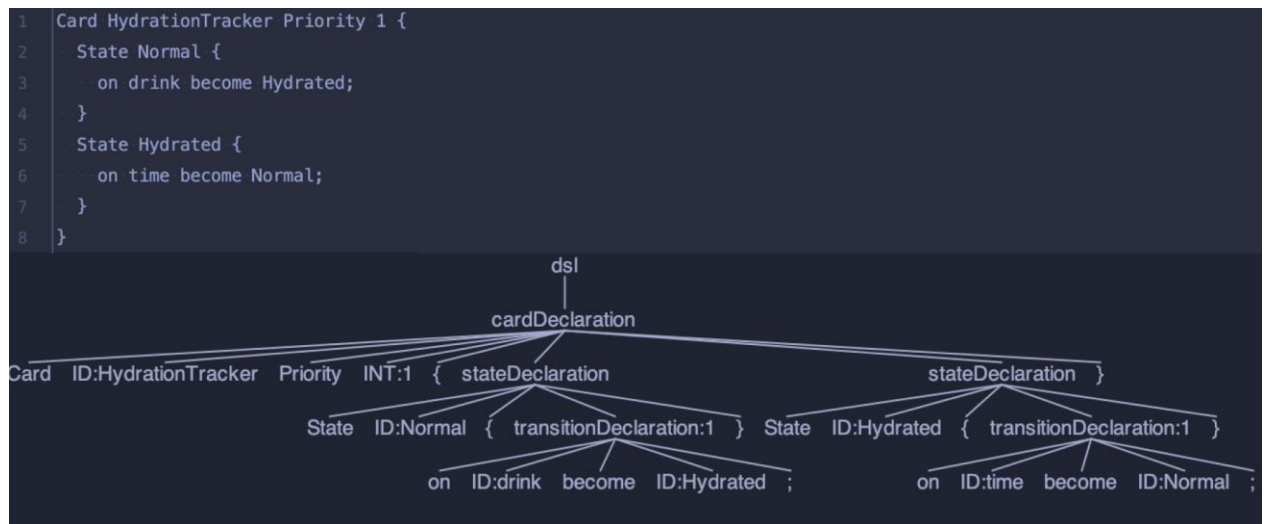


Рисунок 4.2 — приклад коду конфігурації картки та його презентація у вигляді абстрактного синтаксичного дерева

Для інтеграції та обробки даної граматики в Android-додатку, нам необхідно до нашого проекту додати залежність основних класів ANTLR а також конфігурувати генерацію створених на мові Java парсерів, лексера, токенів а також супутніх класів обробки абстрактного синтаксичного дерева: `visitor` та `listener`.

Звертаючись до документації [83] ми визначаємо, що правила граматики можна обробляти двома методами:

1. Обробка правила при вході та при виході. Цей спосіб називається `listener`.
2. Обробка під час входу в нього та подальше повернення значення, яке буде оброблятися правилом, що знаходиться вище за ієрархією. Цей спосіб має назву `visitor`.

Реалізуючі `visitor` або `listener` на мові Kotlin, ми фактично створюємо інтерпретатор для нашої зовнішньої предметно орієнтованої мови, котрий однозначно буде трактувати попередньо узгоджені умови та будувати стійку систему, що має єдине джерело правди у вигляді граматики мови. Таким чином розробники клієнтського додатку будуть отримувати завжди очікувані та передбачувані дані, а сторона замовника отримає формальний інструмент, котрий буде слугувати контрактом між ними та клієнтською стороною додатку.

4.2 Програмний прототип внутрішньої DSL

Іншим варіантом реалізації поставленого теоретичного завдання є реалізація за допомогою мови Kotlin внутрішньої DSL. Ми наводимо спрощений приклад, котрий включає в себе лише визначення необхідних конструкцій створюваної предметно-орієнтованої мови.

Для зручності подальшого порівняння ми створили максимально подібні до створеної в минулому розділі зовнішньої DSL. При проектуванні ми використали всі три описані нами користувацькі патерни створення внутрішньої предметно-орієнтованої мови, а саме: ланцюжок викликів функцій – без якого було б неможливо проводити конфігурацію об'єктів карток, контекстні змінні – за допомогою яких ми обмежували область видимості, що дозволяло нам досягати проектованої граматики, також нами було використано вкладені конструктори, що є основою створення та конфігурації об'єктів-карток.

Говорячи конкретніше про можливості саме мови Kotlin для проектування внутрішніх предметно-орієнтованих мов, нами були використані стандартні інструменти, котрі ми описували у минулих розділах, такі як: лямбда-функції – за допомогою яких у нас і є можливість вибудовувати граматику в декларативному стилі, лямбди з приймачем – котрі дозволяють нам обмежувати використання синтаксичних конструкцій, що є важливою складовою при побудові власної граматики, анотація `@DslMarker`, аналогічно до лямбд з приймачем має на меті обмежити контекст при використанні визначених нами елементів, самі ж елементи створюється за допомогою звичного кожному розробнику об'єктно-орієнтованому механізму визначення класів та їх параметрів для конструктора одразу ж при описі класу, не менш важливим є й механізм інфіксних функцій, за допомогою яких нами описаний синтаксис ключових слів, що відповідають за зміну станів та реактивні дії щодо подій.

```

class Card(val name: String, var priority: Int) {
    val states = mutableListOf<State>()

    fun state(name: String, block: State.() -> Unit) {
        val state = State(name)
        state.block()
        states.add(state)
    }
}

class State(val name: String) {
    val transitions = mutableListOf<Transition>()

    infix fun String.become(nextState: String) {
        transitions.add(Transition(event: this, nextState))
    }

    infix fun String.now(state: String): Condition {
        return Condition(card: this, state)
    }
}

class Transition(val event: String, val nextState: String)

class Condition(val card: String, val state: String)

fun card(name: String, block: Card.() -> Unit): Card {
    val card = Card(name, priority: 0)
    card.block()
    return card
}

infix fun Card.withPriority(priority: Int) {
    this.priority = priority
}

fun cardDemo() {
    listOf(
        card(name: "HydrationTracker") { this: Card
            withPriority(priority: 1)

            state(name: "Normal") { this: State
                "drink" become "Hydrated"
            }

            state(name: "Hydrated") { this: State
                "time" become "Normal"
            }
        },
    )
}

```

Рисунок 4.3 — Опис внутрішньої DSL на мові Kotlin та приклад її використання

Нажаль, можливості мови Kotlin не дозволили нам повноцінно повторити граматику та синтаксис зовнішньої предметної-орієнтованої мови. Так лексему “IS” довелось замінити на “now”, оскільки перша є зарезервованим ключовим словом самої мови. Також нам не вдалось повторити обов’язкове визначення пріоритету в тому ж стилі, що застосовувався в зовнішній DSL.

4.3 Висновки до розділу

В ході роботи над даним розділом завдяки отриманим теоретичним знанням в ході дослідження ключових понять притаманним сфері розробки власної мови, а також аналізу різноманітного інструментарію для створення

власної предметно-орієнтованої мови, нами було визначено вимоги до гіпотетичної проблеми конфігурацій карток, котра мала на меті продемонструвати можливості, що надаються предметно-орієнтованими мовами, після чого нами було створено програмні прототипи предметно-орієнтованих мов для даної задачі. реалізовано два варіанти DSL – зовнішню та внутрішню. Зовнішня – реалізована за допомогою парсер-генератора ANTLR, котрий ми визначали як найоптимальніший для більшості задач пов’язаних з розробкою Android-додатків, внутрішня ж реалізована завдяки можливостям мови Kotlin, що є індустріальним стандартом розробки Android-застосунків. На рисунку 4.4 продемонстрований приклад конфігурацій карток за допомогою зовнішньої та внутрішніх DSL, котрий слугував основою для аналізу реалізованих прототипів.

```

Card WorkoutCard Priority 2 {
  State NotStarted {
    on start become InProgress;
  }
  State InProgress {
    if (StepCounter is Active) become Paused;
    on complete become Completed;
  }
  State Paused {
    if (StepCounter is Inactive) become InProgress;
  }
  State Completed {
    on nextDay become NotStarted;
  }
}

Card DailyGoals Priority 1 {
  State NotAchieved {
    if (WorkoutCard is Completed) become Achieved;
  }
  State Achieved {
    on nextDay become NotAchieved;
  }
}

```

```

card( name: "WorkoutCard") { this: Card
  withPriority( priority: 2)
  state( name: "NotStarted") { this: State
    | "start" become "InProgress"
  }
  state( name: "InProgress") { this: State
    | "StepCounter" now "Active" become "Paused"
    | "complete" become "Completed"
  }
  state( name: "Paused") { this: State
    | "StepCounter" now "Inactive" become "InProgress"
  }
  state( name: "Completed") { this: State
    | "nextDay" become "NotStarted"
  }
},
card( name: "DailyGoals") { this: Card
  withPriority( priority: 1)
  state( name: "NotAchieved") { this: State
    | "WorkoutCard" now "Completed" become "Achieved"
  }
  state( name: "Achieved") { this: State
    | "nextDay" become "NotAchieved"
  }
}

```

Рисунок 4.4 — Приклад написання конфігурацій карток за допомогою зовнішньої DSL та внутрішньої Kotlin

На рисунку 4.5 зображено візуальне відображення застосування даних мов-конфігураторів.

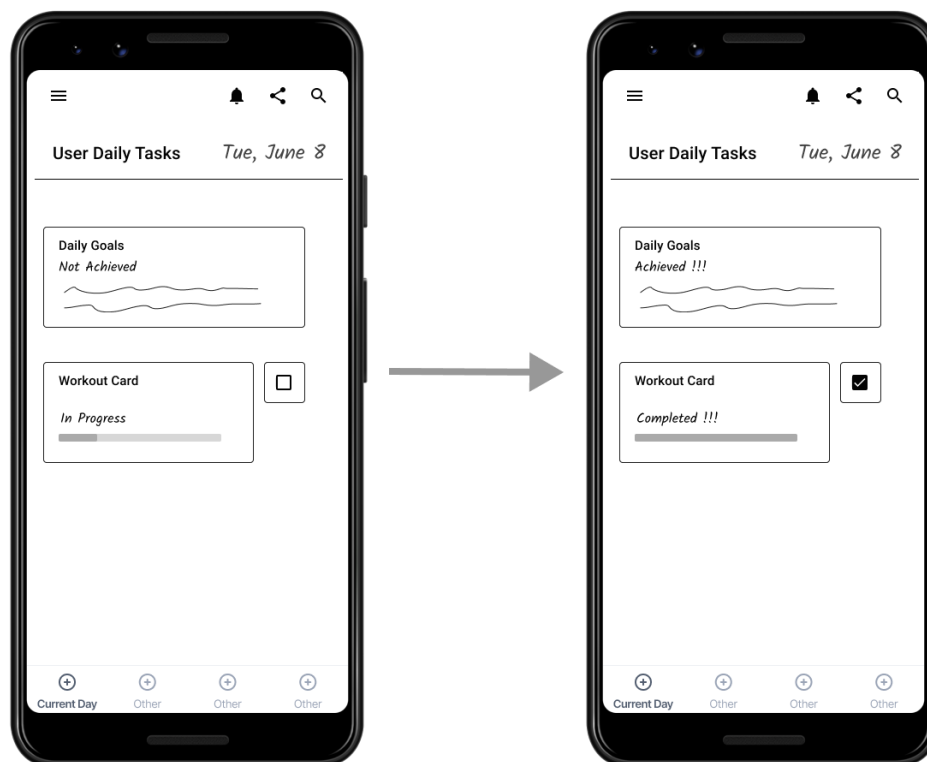


Рисунок 4.5 — візуальне відображення можливого використання створених програмних прототипів

Проаналізувавши реалізовані прототипи, можемо зазначити низку переваг та недоліків кожного з використаних підходів.

До плюсів використання внутрішньої DSL слід віднести простоту реалізації, що полягає в першу чергу з відсутністю необхідності у вивченні зовнішнього інструментарію як-ось ANTLR чи JetBrains MPS, варто зазначити тако безшовність інтеграції в застосунок, завдяки чому розробники можуть зосередитись виключно на проектуванні граматики мови, останнім вагомим плюсом є те, що розробникам не потрібно проектувати граматику мову з нуля, а лише визначити дії та ключові слова, що будуть застосовані для проєкції домену.

До мінусів же ми відносимо обмежену можливість з проектування граматики, що може заважати реалізації специфічних до вимог сценаріїв використання, обмеження накладені мовою програмування щодо ключових слів, відсутність розмежування між лексером, парсером та інтерпретатором, а також високий поріг входу для використання та інтеграції для не-розробників.

До плюсів використання зовнішньої DSL однозначно слід віднести абсолютно необмежені можливості щодо створення та проектування власної граматики, у низці інструментів чітке розділення на реалізацію парсера, лексера та інтерпретатора, що дозволяє створювати архітектурно більш незалежні рішення (наприклад написання інтерпретатора на різних мовах), за допомогою аналізатора синтаксичного дерева можливо створювати редактори для користувачів, що не є розробниками, що спрощує використання ними цієї мови.

До мінусів же ми можемо віднести високий поріг входу для створення та інтеграції такого рішення, а також його вузьку направленість, що й робить даний підхід нажаль не популярним серед спільноти.

Нами було створено дві порівняльні таблиці 4.1 та 4.2, котрі резюмують переваги та недоліки визначені під час реалізації програмних прототипів та аналізу та дослідженню технічних можливостей щодо створення різних видів предметно-орієнтованих мов.

Таблиця 4.1

Переваги та недоліки внутрішньої DSL

Переваги внутрішньої DSL	Недоліки внутрішньої DSL
Простота реалізації	Обмежена можливість проектування граматики
Відсутність необхідності у вивченні зовнішнього інструментарію (наприклад, ANTLR, JetBrains MPS)	Обмеження накладені мовою програмування на ключові слова
Безшовність інтеграції в застосунок	Відсутність розмежування між лексером, парсером та інтерпретатором
Можливість зосередитись на проектуванні граматики	Високий поріг входу для використання та інтеграції для не-розробників
Необхідність лише визначення дій та ключових слів для проекції домену	Відсутність гнучкості в реалізації специфічних сценаріїв використання

Таблиця 4.2

Переваги та недоліки зовнішньої DSL

Переваги зовнішньої DSL	Недоліки зовнішньої DSL
Абсолютно необмежені можливості щодо створення та проектування граматики	Високий поріг входу для створення та інтеграції
Чітке розділення на реалізацію парсера, лексера та інтерпретатора	Вузька направленість, робить підхід не популярним серед спільноти
Можливість створювати архітектурно більш незалежні рішення	
Створення редакторів для користувачів, що не є розробниками	

5 РОЗРОБКА СТАРТАП ПРОЕКТУ

У сучасному світі, де програмування та розробка програмного забезпечення набувають все більшої важливості, ключову роль відіграє ефективність та гнучкість розробки. Однією з інноваційних тенденцій у цій сфері є розробка специфічних мов домену (DSL) для конкретних додатків та платформ. Наш стартап-проект фокусується на створенні та впровадженні DSL для різних Android-продуктів, пропонуючи унікальні та ефективні рішення для програмістів та розробників.

Сфера мобільних технологій є однією з найдинамічніших та найшвидше розвиваючихся галузей у світі. З ростом попиту на інноваційні мобільні додатки, значно збільшується потреба у високоякісних інструментах для їх розробки. Впровадження DSL у процес розробки Android-додатків відкриває нові можливості для оптимізації та автоматизації, що, у свою чергу, сприяє підвищенню продуктивності та якості кінцевих продуктів.

Цей розділ присвячений дослідженню та розробці стартап-проекту, який орієнтований на консультації, створення та впровадження DSL для різноманітних Android-продуктів. Метою цього стартапу є надання комплексних рішень для розробників мобільних додатків, що сприятиме підвищенню продуктивності, зниженню витрат та підвищенню якості кінцевого продукту.

Далі у розділі буде проведено аналіз ринку, розроблено стратегії входу на ринок та маркетингові стратегії, з урахуванням специфіки даного стартап-проекту. Також буде розглянуто ключові переваги використання DSL у контексті Android-розробки, аналіз конкурентів, потенційних ризиків та можливостей, які можуть вплинути на успішність проекту.

5.1 Розробка плану стартапу та його ринкове масштабування

Розглянемо стратегію розвитку стартапу та його вихід на ринок. Спершу потрібно здійснити аналіз ринку, який охоплює:

- вивчення конкурентів для з'ясування існуючих методів вирішення проблем,
- визначення концепції проекту та його цільової аудиторії,
- розробка плану запуску продукту, виходячи з умов ринку.

Потім слід організувати стартап, розробивши повний план розвитку і випуску продукту:

- планування виробничих потужностей та ресурсів;
- розрахунок витрат на реалізацію проекту.

Наступний крок - фінансово-економічний аналіз та оцінка ризиків, включаючи:

- розрахунок інвестиційних витрат,
- розрахунок ключових фінансових показників
- визначення ризиків з можливими шляхами їх уникнення.

Останній етап - розробка стратегії комерціалізації продукту, важливого для його масштабування, який включає:

- пошук інвесторів і різних шляхів фінансування,
- складання інвестиційної пропозиції
- вибір каналів спілкування з потенційними інвесторами.

5.2 Опис ідеї стартап-проекту

Стартап-проект передбачає консультації та розробку інноваційної системи, яка спеціалізується на створенні та впровадженні DSL для різноманітних продуктів на платформі Android.

Проект націлений на полегшення та прискорення процесу розробки програмного забезпечення для мобільних пристроїв, забезпечуючи більш ефективне використання ресурсів і оптимізацію робочого процесу.

Основна мета проекту – створити інструменти, які б дозволили розробникам швидше та якісніше створювати додатки, підвищуючи тим самим продуктивність і конкурентоспроможність на ринку Android-додатків. У таблиці 5.1 нижче ми наводимо інформаційну карту стартапу.

Інформаційна карта стартап-проекту

Назва проекту	DSL Android Assistant
Автори проекту	Токар Михайло Євгенович
Коротка анотація	Стартап-проект з розробки спеціалізованої мови програмування (DSL) для ефективної роботи з Android-додатками, що полегшить процес розробки та інтеграції продуктів.
Термін реалізації проекту	18 місяців
Головні цілі та завдання проекту	Створення і впровадження унікальної DSL для Android-додатків, яка забезпечуватиме легшу та швидшу розробку, інтеграцію та налаштування продуктів.
Необхідні ресурси	Команда розробників зі знанням Android та DSL, обладнані робочі станції, програмне забезпечення для розробки та тестування, фінансування на оплату праці, оренду офісу та маркетингові витрати.
Опис проблеми, яку вирішує проект	Спрощення та оптимізація процесу розробки Android-додатків через впровадження DSL, що знижує час та зусилля на програмування та тестування.
Очікувані результати	Збільшення продуктивності розробників Android-додатків, зменшення часу на впровадження нових продуктів на ринок, залучення уваги та інвестицій від технологічних компаній.

5.3 Технологічний аудит ідеї проекту

Розглянемо концепцію стартапу DSL Android Assistant та виконаємо конкурентний аналіз. У таблиці 5.2 представлений детальний опис ідеї цього проекту.

Представлений продукт, описаний у таблиці 5.2, є інноваційним на ринку і не має прямих аналогів. Однак потенційними конкурентами можуть бути великі компанії та корпорації, здатні фінансувати розробку альтернативних чи більш функціональних продуктів, або компанії, що працюють над створення редакторів предметно-орієнтованих мов програмування та можуть розширити свою діяльність надаючи клієнтам послуги з консультації та впровадження даного виду інструментарію, що потенційно може призвести до появи сильних конкурентів чи поглинання нашого стартапу великою компанією. Проведений аналіз конкурентів і його результати будуть описувати різноманітні бізнес аспекти стартапу та представлені у таблиці 5.3.

Опис ідеї стартапу

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Розробка та впровадження DSL для Android-продуктів	Розробка мобільних додатків на Android	Полегшення процесу розробки, зниження витрат часу та ресурсів

Таблиця 5.3

Порівняльний аналіз конкурентів проекту

Техніко-економічні характеристики	Мій проект	Конкурент
Ціна	\$1500	\$2500
Доступність на ринку	Наявна	Обмежена
Підтримка	1 рік безкоштовно, потім за плату	6 місяців безкоштовно, потім за плату
Програмні ресурси	Використання відкритих та безкоштовних бібліотек	Переважно платні ресурси
Фінансова та технічна залежність	Мінімальна	Висока

Далі аналізуємо технічно спроможність втілення ідеї проекту (таблиця 5.4).

Таблиця 5.4

Здійсненність продукту у технологічному плані

№ п/п	Ідея проекту	Технології і реалізації	Наявність технологій	Доступність технологій
1	Розробка специфічної DSL для Android	Використання Java/Kotlin, DSL технік	Наявні	Доступні

Продовження таблиці 5.4

2	Інтеграція DSL з існуючими Android-додатками	Адаптація DSL під Android SDK та різноманітні бібліотеки	Наявні	Доступні
---	--	--	--------	----------

5.4 Ринкові можливості для запуску стартап-проекту

Ми здійснимо аналіз ринкових умов для ефективного запуску нашого стартапу "DSL Android Assistant", орієнтуючись на відомості, представлені в таблиці 5.5. Наступним кроком буде оцінка ринкових можливостей та загроз, які можуть вплинути на розвиток цього проекту. Цей аналіз допоможе нам визначити оптимальні напрямки розвитку, враховуючи поточну ринкову ситуацію, потреби потенційних клієнтів та стратегії наших конкурентів. Особливу увагу приділимо аналізу попиту на ринку, що відображено в таблиці 5.6.

Таблиця 5.5

Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники ринку (найменування)	Характеристика
1	Кількість головних гравців, од	5
2	Загальний обсяг продаж, грн/ум.од	8000
3	Динаміка ринку (якісна оцінка)	Позитивна, стрімке зростання
4	Наявність обмежень для входу (вказати характер обмежень)	Відсутні
5	Специфічні вимоги до стандартизації та сертифікації	Стандарти безпеки даних
6	Специфічні вимоги до стандартизації та сертифікації	15%

Виходячи з результатів аналізу, ІТ ринок в Україні незважаючи на обтяжуючі соціально-економічні обставини стрімко розвивається та має широку інвестиційну привабливість.

Проведемо аналіз характеристики, щодо залучення потенційних клієнтів, котрі можуть проявляти певну зацікавленість до проекту у таблиці 5.6.

Таблиця 5.6

Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних груп клієнтів	Вимоги споживачів до товару
Підвищення ефективності та швидкості розробки Android-додатків	Розробники мобільних додатків, ІТ-компанії, стартапи у сфері технологій	Різниця в технічних потребах та ресурсах для впровадження	Наявність сертифікатів якості, патентів

Обрахуємо різноманітні екзистенційні фактори у таблицях 5.7 та 5.8, серед яких є загрози та можливості. Проаналізуємо першу категорію факторів задля розуміння перешкод, котрі стають на заваді запуску продукту на ринок. В свою чергу, можливості необхідно обрахувати для визначення сприятливих умов, котрими потенційно можливо скористатися.

Таблиця 5.7

Фактори загроз

Фактор	Зміст загрози	Можлива реакція компанії
Технологічні зміни	Швидке оновлення технологій може зробити продукт застарілим	Постійне оновлення та адаптація до новітніх технологій
Конкуренція у сфері розробки ПЗ	Поява нових технологій та рішень від конкурентів	Інновації та унікальність продукту, акцент на якості та підтримці
Зміни у законодавстві	Нові норми щодо програмування та захисту даних	Слідкування за законодавчими змінами та швидка адаптація

Фактори можливостей

Фактор	Зміст можливості	Можлива реакція компанії
Збільшення попиту на спеціалізовані DSL	Розвиток ринку спеціалізованих DSL	Активний маркетинг і розширення продуктової лінійки
Глобалізація ринку ПЗ	Вихід на міжнародні ринки	Локалізація продукту, створення міжнародних партнерств
Інновації у сфері розробки	Потреба у нових рішеннях для розробників	Фокус на дослідженнях та розвитку інноваційних функцій

Далі розглянемо питання конкуренції, а саме визначимо її тип та рівень (таблиця 5.9).

Таблиця 5.9

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	У чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Тип конкуренції: Високотехнологічна	Новизна і технологічна складність	Інвестиції у дослідження та розробку
2. Рівень конкурентної боротьби: Глобальний	Присутність на міжнародних ринках	Стратегія глобалізації та адаптації
3. Галузева ознака: Технологічна	Спеціалізація на конкретному сегменті ПЗ	Фокус на глибокій спеціалізації
4. Конкуренція за видами товарів: Спеціалізоване ПЗ	Конкуренція з іншими DSL і Android-інструментами	Розробка унікальних функцій і можливостей
5. Характер конкурентних переваг: Технологічний	Розробка інноваційних рішень	Постійне оновлення та покращення продукту
6. Інтенсивність конкуренції: Висока	Наявність сильних конкурентів	Створення власного бренду і маркетингових стратегій

Наступним етапом ми маємо створити аналіз конкуренції відповідно до моделі 5 сил конкуренції авторства Майкла Портера (таблиця 5.10).

Таблиця 5.10

Аналіз конкуренції в галузі за М. Портером

	Прямі конкуренти у галузі	Потенційні конкуренти	Постачальники	Клієнти	Товарозамінники
Складові аналізу	Інші розробники DSL і платформ для Android	Нові стартапи та компанії, які можуть входити на ринок з аналогічними рішеннями	Оскільки продукт є програмним рішенням, постачальників як таких немає	Розробники мобільних додатків, ІТ-компанії, стартапи	Традиційні мови програмування та інструменти розробки для Android
Висновки	Продукт знаходиться в умовах помірної конкуренції з можливістю розвитку через унікальність та інноваційність. Є потенціал для приваблення нових клієнтів, хоча на ринку присутні товарозамінники.				

Наразі, нами зроблено аналіз конкуренції, котрий демонструється у таблиці 5.10, ідеї характеристик нашого стартап-проекту продемонстровано у таблиці 5.5, аналіз потенційних клієнтів, а також їхніх теоретичних вимог поставлених до продукту визначено у таблиці 5.6. Фактори, що притаманні ринковому середовищу, а також сформульований перелік факторів, котрі впливають на показники конкурентоспроможності продемонстровано у таблицях 5.7 і 5.8 та 5.11.

Таблиця 5.11

Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1	Унікальність технології	DSL для Android надає конкурентну перевагу завдяки інноваційному підходу
2	Інтеграція з існуючими системами	Можливість легкої інтеграції з існуючими Android-додатками
3	Висока продуктивність	Підвищення швидкості розробки і зниження витрат часу для користувачів
4	Гнучкість та адаптивність	Можливість налаштування під різні потреби розробників

Далі нами буде проведено аналіз сильних та слабких сторін нашого продукту, результати демонструються у таблиці 5.12.

Таблиця 5.12

Порівняльний аналіз сильних та слабких сторін системи

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг конкурентів							
			-3	-2	-1	0	1	2	3	
1	Унікальність технології	18	+							
2	Інтеграція	16			+					
3	Продуктивність	17		+						
4	Гнучкість	15				+				

Далі нам необхідно провести SWOT-аналіз.

Таблиця 5.13

SWOT-аналіз стартап-проекту

Сильні сторони Інноваційність, гнучкість	Слабкі сторони Новизна на ринку, відсутність визнання
Можливості Розвиток ринку DSL, нові партнерства	Загрози Конкуренція, технологічні зміни

Проведення SWOT-аналізу дозволило нам ідентифікувати ключові сильні та слабкі сторони проекту, а також виявити потенційні можливості та загрози,

що мають важливе значення для стратегічного планування та конкурентної боротьби. На основі цих даних ми розробимо стратегію ринкової поведінки, щоб ефективно інтегрувати стартап на ринок, враховуючи можливі майбутні напрями розвитку та часові рамки впровадження продукту, результати яких будуть представлені в наступних таблицях, як ось таблиці 5.14.

Таблиця 5.14

Альтернативи ринкового впровадження стартап проекту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Швидкий запуск з базовими функціями	80%	3 місяці
2	Повільний запуск з розширеними можливостями	60%	8 місяців
3	Фокус на спеціалізованих сегментах	70%	6 місяців

У даному пункті був проведений детальний аналіз ринку та продукту. Також відповідно до результатів проведеного конкурентного аналізу, визначених факторів ринку та його сприятливості, описання ідеї та характеристик стартап-проекту, робимо висновок, що існують дуже сприятливі умови для виходу продукту на ринок.

5.5 Розробка ринкової стратегії стартап-проекту

В контексті даного підрозділу для розробки стратегії продукту, нами було проаналізовано цільову аудиторію проекту, представлено у вигляді таблиці 5.15 та базові стратегії щодо розвитку продукту у вигляді таблиці 5.16.

Таблиця 5.15

Вибір цільових груп потенційних споживачів

п/п	Опис профілю цільової групи	Готовність до сприйняття	Орієнтовний попит	Інтенсивність конкуренції	Простота входу
	Розробники мобільних додатків	Висока	30%	Висока	Середня
	ІТ-компанії, стартапи	Висока	35%	Висока	Середня

Продовження таблиці 5.15

Незалежні програмісти	Середня	25%	Середня	Низька
Які цільові групи обрано: 1, 2				

Таблиця 5.16

Визначення базової стратегії розвитку

№ п/п	Обрана альтернатива	Стратегія охоплення	Ключові позиції	Базова стратегія
1	1 та 2	Диференційований маркетинг	Інновації, адаптивність	Розвиток та інновації

В обраних нами сегментах ринку нам було сформовано базову для роботи стратегію щодо розвитку у вигляді таблиці 5.17 та 5.18.

Таблиця 5.17

Визначення базової стратегії конкурентної поведінки

Першопрохідство	Пошук споживачів	Копіювання характеристик	Стратегія поведінки
Так	Шукати нових	Ні	Інноваційний лідер

Таблиця 5.18

Визначення стратегії позиціонування

Вимоги аудиторії	Базова стратегія	Ключові позиції	Асоціації
Універсальність, простота, якість	Інновації, оптимальні витрати	Гнучкість, простота, висока якість	Інноваційність, ефективність, надійність

5.6 Розробка маркетингової програми стартап-проекту

Після проведеного комплексного аналізу, можемо повноцінно описати ключові переваги концепції потенційного товару (таблиця 5.19) та побудувати концепцію маркетингових комунікацій (таблиця 5.20).

Таблиця 5.19

Ключові переваги концепції потенційного товару

№ п/п	Потреба	Вигода	Ключові переваги
1	Ефективність розробки	Збільшення продуктивності	Унікальність технології, висока адаптивність
2	Гнучкість використання	Підходить для різних проектів	Можливість налаштування під конкретні потреби
3	Простота інтеграції	Легке впровадження у вже існуючі системи	Мінімізація часу на впровадження та навчання

Таблиця 5.20

Концепція маркетингових комунікацій

№ п/п	Поведінка клієнтів	Канали комунікації	Ключові позиції	Завдання повідомлення	Концепція рекламного звернення
1	Інтерес до нових технологій	Соціальні мережі, технічні форуми, вебіари	Інновації, ефективність	Підкреслити унікальність та ефективність продукту	Розробка як інструмент досягнення кращих результатів
2	Пошук ефективних рішень	Блоги, YouTube-канали, IT-конференції	Простота, надійність	Показати легкість використання та надійність	Продукт, який робить розробку простішою та надійнішою
3	Прагнення до оптимізації витрат	Цільова реклама, електронні листи	Вартість, оптимізація	Висвітлити економію часу та ресурсів	Ефективне вирішення задач за менші витрати

5.7 Висновки до розділу

Цей розділ презентує дослідження стартапу на основі предмету дослідження наукової роботи. Стартап проект "DSL Android Assistant" – спеціалізується на консультації, розробці та підтримці застосування спеціалізованої мови програмування в контексті роботи різноманітних Android-додатків.

Розглянуто стратегії виходу на ринок та маркетингові підходи. Ринок для цього продукту характеризується обмеженою конкуренцією та великим потенціалом для універсальної, доступної системи, що відкриває шляхи для домінування стартапу.

Висвітлено сильні та слабкі сторони проекту, проведено SWOT-аналіз та аналіз цільової аудиторії. На підставі цих даних сформульована маркетингова стратегія для вибраних цільових груп.

ВИСНОВКИ

Відповідно до мети даної роботи ми визначили та виконали завдання, що були сформульовані для її досягнення. Нами було досліджено теоретичний матеріал необхідний для розуміння процесу проектування предметно-орієнтованих мов, за однією з класифікаційних ознак виокремлено два типи DSL внутрішні, що пишуться на основі хост-мови та зовнішні, що створюється незалежно. Нами було досліджено можливості основної мови програмування Android-додатків Kotlin для створення внутрішніх DSL. В свою чергу для зовнішніх DSL виокремлено три форми мови: текстова, графічна та проекційна. Проаналізовано сучасний інструментарій для створення кожного типу мови, та досліджено можливості інструментів щодо застосування та інтеграції у розробку клієнтських застосунків для платформи Android. Визначено найбільш оптимальні інструментами для поставлених завдань: парсер-генератор текстових мов ANTLR та проекційний редактор JetBrains MPS. Обидва інструменти мають високі показники щодо можливості інтеграції в Android додатки завдяки можливості визначати поведінку спроектованої мови за допомогою мови програмування Kotlin.

Внутрішні DSL мають широке застосування в контексті Android-розробки в доменах створення користувацького інтерфейсу, тестування, конфігурації правил проектної збірки, ін'єкції залежностей тощо. Зовнішні ж DSL застосовуються аналогічно при створенні користувацького інтерфейсу, високопродуктивних елементів інтерфейсу, для опису запитів до баз даних та специфічних сценаріїв використання окремими виробництвами.

Для аналізу ефективності та придатності кожного підходу були створені вимоги до програмного продукту для опису карток користувацького інтерфейсу. За допомогою парсер-генератора ANTLR було описано граматику мови, створено лексер та парсер, та протестована інтеграція інтерпритатора створюваної мови до Android-додатку. За допомогою ж інструментів мови програмування Kotlin було створено аналогічну за можливостями, та подібною

за синтаксисом внутрішню DSL. В ході аналізу та порівняння даних підходів ми дійшли до наступного висновку: плюси внутрішньої DSL включають її простоту у реалізації, зокрема завдяки відсутності необхідності вивчати додаткові зовнішні інструменти, легкість інтеграції в застосунок, та можливість використання існуючих граматики мови. Однак, цей підхід має обмеження в проектуванні граматики та високий поріг написання коду для не-розробників. З іншого боку, зовнішня DSL надає більшу свободу у створенні граматики та можливість створювати незалежні архітектурні рішення, а також редактори для користувачів-непрограмістів. Однак, вона вимагає вищого рівня вмінь для створення та інтеграції, а її специфічність знижує популярність серед розробників.

Виконавши всі поставлені в роботі завдання ми можемо зазначити, що дослідження в даній тематиці мають високу перспективу розвитку у вигляді проектування більш складної предметної логіки та докладнішому опису процесу написання інтерпритатора, а також акценті на використанні принципів метапрограмування.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

- [1] American Standard Vocabulary for Information Processing, X3.12, American Standards Association [now United States of America Standards Institute], New York, 1966..
- [2] IFIP-ICC Vocabulary of Information Processing (First English language edition). North-Holland Publishing Co., Amsterdam, Netherlands, 1966..
- [3] "Bjarne Stroustrup, The Design and Evolution of C++," [Online]. Available: http://www.cs.umbc.edu/331/papers/dne_notes.pdf. [Accessed 1 1 2023].
- [4] Maurizio Gabbrielli and Simone Martini, Programming Languages: Principles and Paradigms, DOI 10.1007/978-1-84882-914-5, Springer London Dordrecht Heidelberg New York.
- [5] Інформаційні системи і технології у фінансах, конспект лекцій / Укладач В.М.Олійник.– Суми: Вид-во СумДУ, 2010.– 172с..
- [6] Страхування : навч. посіб. / Г. М. Рябенко, Н. М. Сіренко, А. С. Кравченко — Миколаїв : МНАУ, 2014. — 429 с...
- [7] "Важливість Рівня Абстракції," [Online]. Available: <https://dou.ua/lenta/articles/level-of-abstraction/>. [Accessed 1 1 2024].
- [8] Programming Paradigms for Dummies What Every Programmer Should Know, Peter Van Roy, Université Catholique de Louvain - UCLouvain.
- [9] Peter Van-Roy; Seif Haridi (2004). Concepts, Techniques, and Models of Computer Programming. MIT Press. ISBN 978-0-262-22069-9..
- [10] "Парадигми програмування. Елементи програм C++," [Online]. Available: http://om.univ.kiev.ua/users_upload/15/upload/file/prog_lecture_01.pdf. [Accessed 1 1 2024].

- [11] Lorenzo Bettini. Implementing Domain-Specific Languages with Xtext and Xtend. Published by Packt Publishing Ltd. Livery Place 35 Livery Street Birmingham B3 2PB, UK..
- [12] A preliminary study on various implementation approaches of domain-specific language, Tomaz̃ Kosar, Pablo E. Martinez Lopez, a University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, 2000 Maribor, Slovenia, 2007.
- [13] DSL Engineering Designing, Implementing and Using Domain-Specific Languages, Markus Voelter, (c) 2010 - 2013 Markus Voelte.
- [14] DSLs in Action by Debasish Ghosh Released November 2010 Publisher(s): Manning Publications ISBN: 9781935182450.
- [15] Domain Specific Languages By: Martin Fowler Publisher: Addison-Wesley Professional Pub. Date: September 24, 2010 Print ISBN-10: 0-321-71294-3.
- [16] Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, Sebastian Günther, Sebastian Günther, Software Languages Lab, Faculty of Sciences, Vrije Universiteit Brussel, Pleinlaan 2.
- [17] Domain-Specific Languages: An Annotated Bibliography, Arie van Deursen, CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherland.
- [18] "The complete guide to (external) Domain Specific Languages," [Online]. Available: <https://tomasseti.me/domain-specific-languages/>. [Accessed 1 1 2024].
- [19] "An alternative to SourceEditing - Projectional Editing," [Online]. Available: <https://martinfowler.com/bliki/ProjectionalEditing.html>. [Accessed 1 1 2024].
- [20] Гаврилків В. М. Формальні мови та алгоритмічні моделі. — І.-Ф. : Голіней, 2023. — 180 с..
- [21] Frost, R., Hafiz, R. and Callaghan, P. (2007) « Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars .» 10th

- International Workshop on Parsing Technologies (IWPT), ACL-SIGPARSE , Pages: 109—120, June 2007, Prague..
- [22] Abstract Syntax Tree Design, Nicola Howarth, APM.1551.01, Poseidon House Castle Park Cambridge CB3 0RD United Kingdom, 23rd August 1995.
- [23] "Abstract Syntax Tree Unparsing - Available Kinds of Unparser," [Online]. Available: https://web.archive.org/web/20070913222016/http://eli-project.sourceforge.net/elionline4.4/idem_2.html. [Accessed 11 2024].
- [24] ↑ Frost, R., Hafiz, R. and Callaghan, P. (2008) « Parser Combinators for Ambiguous Left-Recursive Grammars.» 10th International Symposium on Practical Aspects of Declarative Languages (PADL), ACM-SIGPLAN , Volume 4902/2008, Pages: 167—181, January 2008, S.
- [25] Change Distilling: Tree Differencing for FineGrained Source Code Change Extraction, Beat Fluri, Student Member, IEEE, Michael Wursch, published online 3 Aug. 2007, IEEECS Log Number TSE-0012-0107..
- [26] How to create pragmatic, lightweight languages - Learn the process to create DSLs and GPLs, Federico Tomassetti, Leanpub 2020, 370 pages..
- [27] "ANTLR Website," [Online]. Available: <https://www.antlr.org/>. [Accessed 11 2024].
- [28] The Definitive ANTLR 4 Reference, Copyright Terrence Parr, © 2012 The Pragmatic Programmers, LLC, Dallas, Texas, ISBN-13: 978-1-93435-699-9, September 19, 2012.
- [29] Ritchie, Dennis M. (April 1993). "The Development of the C Language". History of programming languages---II. Association for Computing Machinery, Inc. (published 1996-01-01). doi:10.1145/234286.1057834. ISBN 0-201-89502-1. pp. 675, 684:.
- [30] Yacc: Yet Another Compiler-Compiler, Stephen C. Johnson, PS1:15-2, June 1975.

- [31] Levine, John R.; Mason, Tony; Brown, Doug (1992). *lex & yacc* (2 ed.). O'Reilly. pp. 1–2. ISBN 1-56592-000-7..
- [32] Levine, John (August 2009). *flex & bison*. O'Reilly Media. p. 304. ISBN 978-0-596-15597-1..
- [33] Lesk, M.E. (October 1975). "Lex – A Lexical Analyzer Generator". *Comp. Sci. Tech. Rep. No. 39*. Murray Hill, New Jersey: Bell Laboratories.
- [34] Lesk, M.E. (October 1975). "Lex – A Lexical Analyzer Generator". *Comp. Sci. Tech. Rep. No. 39*. Murray Hill, New Jersey: Bell Laboratories..
- [35] The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008) Copyright © 2001-2018 IEEE and The Open Group.
- [36] "PLY (Python Lex-Yacc)," [Online]. Available: <https://www.dabeaz.com/ply/index.html>. [Accessed 1 1 2024].
- [37] "ANTLR Showcase List," [Online]. Available: <https://web.archive.org/web/20080117104239/http://antlr.org/showcase/list>. [Accessed 1 1 2024].
- [38] "ANTLR v4 - IntelliJ IDEs Plugin | Marketplace," [Online]. Available: <https://plugins.jetbrains.com/plugin/7358-antlr-v4>. [Accessed 1 1 2024].
- [39] "ANTLR4 grammar syntax support - Visual Studio Marketplace," [Online]. Available: <https://marketplace.visualstudio.com/items?itemName=mike-lichke.vscode-antlr4>. [Accessed 1 1 2024].
- [40] "The website of Eclipse Xtext, an open-source framework for development of programming languages and domain-specific languages," [Online]. Available: <https://eclipse.dev/Xtext/>. [Accessed 1 1 2024].
- [41] "Eclipse Modeling Project | The Eclipse Foundation," [Online]. Available: <https://eclipse.dev/modeling/emf/>. [Accessed 1 1 2024].
- [42] "Xtext - Platform Comprasion," [Online]. Available: <https://eclipse.dev/Xtext/#platform-comparison>. [Accessed 1 1 2024].

- [43] "textX PyPI Meta-language for DSL implementation inspired by Xtext," [Online]. Available: <https://pypi.org/project/textX/>. [Accessed 1 1 2024].
- [44] "GitHub - textX/textX: Domain-Specific Languages and parsers in Python made easy," [Online]. Available: <https://github.com/textX/textX>. [Accessed 1 1 2024].
- [45] "Spoofox: The Language Designer's Workbench - Spoofox," [Online]. Available: <https://spoofox.dev/>. [Accessed 1 1 2024].
- [46] "Graphical Modeling Framework | The Eclipse Foundation," [Online]. Available: <https://eclipse.dev/modeling/gmp/>. [Accessed 1 1 2024].
- [47] "Eclipse GMF copy-paste problem," [Online]. Available: <https://esalagea.wordpress.com/2011/04/13/lets-solve-once-for-all-the-gmf-copy-paste-problem-and-then-forget-about-it/>. [Accessed 1 1 2024].
- [48] "Eugenia - Epsilon," [Online]. Available: <https://eclipse.dev/epsilon/doc/eugenia/>. [Accessed 1 1 2024].
- [49] "Sirius | Overview," [Online]. Available: <https://eclipse.dev/sirius/overview.html>. [Accessed 1 1 2024].
- [50] "MetaEdit+ Domain-Specific Modeling tools," [Online]. Available: <https://www.metacase.com/products.html>. [Accessed 1 1 2024].
- [51] "MPS: The Domain-Specific Language Creator by JetBrains," [Online]. Available: <https://www.jetbrains.com/mps/>. [Accessed 1 1 2024].
- [52] "LionWeb - GitHub," [Online]. Available: <https://github.com/LionWeb-io>. [Accessed 1 1 2024].
- [53] Building Domain Specific Language in Kotlin, Greg Milllete, 2019-11-05, Leanpub.
- [54] DOMAIN-SPECIFIC LANGUAGES IN KOTLIN AND SCALA - A COMPARISON, JOHANNES KEPLER UNIVERSITY LINZ Altenberger Straße 69 4040 Linz, Austria jku.at, March 2022.

- [55] "Kotlin Docs | Kotlin Documentation," [Online]. Available: <https://kotlinlang.org/docs/home.html>. [Accessed 1 1 2024].
- [56] "Kotlin on Android. Now official," [Online]. Available: <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/>. [Accessed 1 1 2024].
- [57] ANDROIDS THE TEAM THAT BUILT THE ANDROID OPERATING SYSTEM, CHET HAASE, ISBN 978-1-7373548-2-6, 2021.
- [58] "JavaFX," [Online]. Available: <https://openjfx.io/>. [Accessed 1 1 2024].
- [59] "React," [Online]. Available: <https://react.dev/>. [Accessed 1 1 2024].
- [60] "GitHub - Kotlin/anko: Pleasant Android application development," [Online]. Available: <https://github.com/Kotlin/anko>. [Accessed 1 1 2024].
- [61] "GitHub - LouisCAD/Splitties: A collection of hand-crafted extensions for your Kotlin projects.," [Online]. Available: <https://github.com/LouisCAD/Splitties>. [Accessed 1 1 2024].
- [62] "GitHub - Kotlin/kandy: Kotlin plotting library.," [Online]. Available: <https://github.com/Kotlin/kandy>. [Accessed 1 1 2024].
- [63] "Espresso | Android Developers," [Online]. Available: <https://developer.android.com/training/testing/espresso>. [Accessed 1 12 2023].
- [64] "GitHub - codecentric/androidtestktx: Kotlin DSL for Espresso and UIAutomator.," [Online]. Available: <https://github.com/codecentric/androidtestktx>. [Accessed 1 1 2024].
- [65] "GitHub - KakaoCup/Compose: Nice and simple DSL for Espresso Compose UI testing in Kotlin," [Online]. Available: <https://github.com/KakaoCup/Compose>. [Accessed 1 1 2024].
- [66] UNIT ТЕСТУВАННЯ ЯК СПОСІБ ВИЯВИТИ СЛАБКІ, О.О. Бородіна, А.О. Горошко, Полтавський національний технічний університет імені Юрія Кондратюка, ПолтНТУ, 2018..

- [67] "MockK | mocking library for Kotlin," [Online]. Available: <https://mockk.io/>. [Accessed 2 1 2024].
- [68] "Download Android Studio & App Tools - Android Developers," [Online]. Available: <https://developer.android.com/studio>. [Accessed 1 1 2024].
- [69] BUILD AUTOMATION TOOLS FOR SOFTWARE DEVELOPMENT A COMPARATIVE STUDY BETWEEN MAVEN, GRADLE AND BAZEL, Mridula Prakash, L&T Technology Services, CTO Office, Mysore, India, DOI: 10.5121/csit.2022.120608.
- [70] "The Apache Groovy programming language," [Online]. Available: <https://groovy-lang.org/>. [Accessed 1 1 2024].
- [71] Building and Testing with Gradle, Tim Berglund and Matthew McCullough, Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472..
- [72] "Koin - The pragmatic Kotlin Injection Framework - developed by Kotzilla and its open-source contributors," [Online]. Available: <https://insert-koin.io/>. [Accessed 1 1 2024].
- [73] "GitHub - kosi-libs/Kodein: Painless Kotlin Dependency Injection," [Online]. Available: <https://github.com/kosi-libs/Kodein>. [Accessed 1 1 2024].
- [74] "Ktor: Build Asynchronous Servers and Clients in Kotlin," [Online]. Available: <https://ktor.io/>. [Accessed 1 1 2024].
- [75] "Layouts in Views | Android Developers," [Online]. Available: <https://developer.android.com/develop/ui/views/layout/declaring-layout>. [Accessed 1 1 2024].
- [76] "Save data using SQLite | Android Developers," [Online]. Available: <https://developer.android.com/training/data-storage/sqlite>. [Accessed 1 1 2024].

- [77] "SQLite Home Page," [Online]. Available: <https://www.sqlite.org/index.html>. [Accessed 1 1 2024].
- [78] "The OpenGL ES Shading Language Spec," [Online]. Available: https://registry.khronos.org/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.pdf. [Accessed 1 1 2024].
- [79] Learning to Understand Image Blur, Shanghang Zhang, Carnegie Mellon University.
- [80] "Shader based render effects in Compose Desktop with Skia," [Online]. Available: <https://www.pushing-pixels.org/2022/04/09/shader-based-render-effects-in-compose-desktop-with-skia.html>. [Accessed 1 1 2024].
- [81] "About Skia | Skia," [Online]. Available: <https://skia.org/about/>. [Accessed 1 1 2024].
- [82] "GitHub - chrisbanes/haze: A blurring modifier for Compose," [Online]. Available: <https://github.com/chrisbanes/haze/blob/main/haze%2Fsrc%2FskikoMain%2Fkotlin%2Fdev%2Fchrisbanes%2Fhaze%2FHazeNode.kt>. [Accessed 1 1 2024].
- [83] "ANTLR Parser Details," [Online]. Available: <https://github.com/antlr/antlr4/blob/master/doc/index.md>. [Accessed 1 1 2024].

ДОДАТКИ

Додаток А

KotlinCardDsl.kt

```
class Card(val name: String, var priority: Int) {  
    val states = mutableListOf<State>()  
  
    fun state(name: String, block: State.() -> Unit) {  
        val state = State(name)  
        state.block()  
        states.add(state)  
    }  
}
```

```
class State(val name: String) {  
    val transitions = mutableListOf<Transition>()  
  
    infix fun String.become(nextState: String) {  
        transitions.add(Transition(this, nextState))  
    }  
  
    infix fun String.now(state: String): Condition {  
        return Condition(this, state)  
    }  
}
```

```
class Transition(val event: String, val nextState: String)
```

```
class Condition(val card: String, val state: String)
```

```
fun card(name: String, block: Card.() -> Unit): Card {  
    val card = Card(name, 0) // Default priority is 0, can be adjusted later  
    card.block()  
    return card  
}
```

```
infix fun Card.withPriority(priority: Int) {  
    this.priority = priority  
}
```

```
fun main() {  
    val cards = mutableListOf(  
        card("StepCounter") {  
            withPriority(3)  
            state("Inactive") {  
                "step" become "Active"  
            }  
            state("Active") {  
                "stop" become "Inactive"  
            }  
        },  
        card("HydrationTracker") {  
            withPriority(4)  
            state("Normal") {  
                "drink" become "Hydrated"  
            }  
            state("Hydrated") {  
                "time" become "Normal"  
            }  
        },  
    )  
}
```

```

card("WorkoutCard") {
  withPriority(2)
  state("NotStarted") {
    "start" become "InProgress"
  }
  state("InProgress") {
    "StepCounter" now "Active" become "Paused"
    "complete" become "Completed"
  }
  state("Paused") {
    "StepCounter" now "Inactive" become "InProgress"
  }
  state("Completed") {
    "nextDay" become "NotStarted"
  }
},
card("DailyGoals") {
  withPriority(1)
  state("NotAchieved") {
    "WorkoutCard" now "Completed" become "Achieved"
  }
  state("Achieved") {
    "nextDay" become "NotAchieved"
  }
}
)
}

```

CardStateDslGrammar.g4

grammar CardStateDslGrammar;

```

// Головний блок для визначення DSL
dsl: cardDeclaration+;

// Блок для визначення карток
cardDeclaration: 'Card' ID 'Priority' INT '{' stateDeclaration+ '}';

// Блок для визначення станів картки
stateDeclaration: 'State' ID '{' transitionDeclaration+ '}';

// Блок для визначення переходів між станами картки
transitionDeclaration: 'on' ID 'become' ID ';'
    | 'if' '(' expression ')' 'become' ID ';';

// Визначення логічних виразів для умовних переходів
expression: ID 'is' ID;

// Термінали для ідентифікаторів та чисел
ID: [a-zA-Z]+;
INT: [0-9]+;

// Ігнорування пробілів та переводів рядків
WS: [ \t\r\n]+ -> skip;

CardsDemo.tcards
Card StepCounter Priority 3 {
    State Inactive {
        on step become Active;
    }
    State Active {
        on stop become Inactive;
    }
}

```

```
}
```

```
Card HydrationTracker Priority 4 {
```

```
  State Normal {
```

```
    on drink become Hydrated;
```

```
  }
```

```
  State Hydrated {
```

```
    on time become Normal;
```

```
  }
```

```
}
```

```
Card WorkoutCard Priority 2 {
```

```
  State NotStarted {
```

```
    on start become InProgress;
```

```
  }
```

```
  State InProgress {
```

```
    if (StepCounter is Active) become Paused;
```

```
    on complete become Completed;
```

```
  }
```

```
  State Paused {
```

```
    if (StepCounter is Inactive) become InProgress;
```

```
  }
```

```
  State Completed {
```

```
    on nextDay become NotStarted;
```

```
  }
```

```
}
```

```
Card DailyGoals Priority 1 {
```

```
  State NotAchieved {
```

```
    if (WorkoutCard is Completed) become Achieved;
```

```

    }
    State Achieved {
        on nextDay become NotAchieved;
    }
}

// Generated from
/Users/mykhailo/Documents/dsl_demo/app/src/main/java/com/mkhtk/dsl_demo/antlr/
CardStateDslGrammar.g4 by ANTLR 4.13.1
package com.mkhtk.antlr;
import org.antlr.v4.runtime.tree.ParseTreeVisitor;

/**
 * This interface defines a complete generic visitor for a parse tree produced
 * by {@link CardStateDslGrammarParser}.
 *
 * @param <T> The return type of the visit operation. Use {@link Void} for
 * operations with no return type.
 */
public interface CardStateDslGrammarVisitor<T> extends
ParseTreeVisitor<T> {
    /**
     * Visit a parse tree produced by {@link
CardStateDslGrammarParser#dsl}.
     * @param ctx the parse tree
     * @return the visitor result
     */
    T visitDsl(CardStateDslGrammarParser.DslContext ctx);
}

```

```

    * Visit a parse tree produced by { @link
CardStateDslGrammarParser#cardDeclaration }.

```

```

    * @param ctx the parse tree

```

```

    * @return the visitor result

```

```

    */

```

```

T

```

```

visitCardDeclaration(CardStateDslGrammarParser.CardDeclarationContext ctx);

```

```

/**

```

```

    * Visit a parse tree produced by { @link
CardStateDslGrammarParser#stateDeclaration }.

```

```

    * @param ctx the parse tree

```

```

    * @return the visitor result

```

```

    */

```

```

T

```

```

visitStateDeclaration(CardStateDslGrammarParser.StateDeclarationContext ctx);

```

```

/**

```

```

    * Visit a parse tree produced by { @link
CardStateDslGrammarParser#transitionDeclaration }.

```

```

    * @param ctx the parse tree

```

```

    * @return the visitor result

```

```

    */

```

```

T

```

```

visitTransitionDeclaration(CardStateDslGrammarParser.TransitionDeclarationContext ctx);

```

```

/**

```

```

    * Visit a parse tree produced by { @link
CardStateDslGrammarParser#expression }.

```

```

    * @param ctx the parse tree

```

```

    * @return the visitor result

```

```

    */

```

```

        T visitExpression(CardStateDslGrammarParser.ExpressionContext ctx);
    }
    // Generated from
/Users/mykhailo/Documents/dsl_demo/app/src/main/java/com/mkhtk/dsl_demo/antlr/
CardStateDslGrammar.g4 by ANTLR 4.13.1
    package com.mkhtk.antlr;
    import org.antlr.v4.runtime.tree.ParseTreeListener;

    /**
     * This interface defines a complete listener for a parse tree produced by
     * {@link CardStateDslGrammarParser}.
     */
    public interface CardStateDslGrammarListener extends ParseTreeListener {
        /**
         * Enter a parse tree produced by {@link
CardStateDslGrammarParser#dsl}.
         * @param ctx the parse tree
         */
        void enterDsl(CardStateDslGrammarParser.DslContext ctx);
        /**
         * Exit a parse tree produced by {@link
CardStateDslGrammarParser#dsl}.
         * @param ctx the parse tree
         */
        void exitDsl(CardStateDslGrammarParser.DslContext ctx);
        /**
         * Enter a parse tree produced by {@link
CardStateDslGrammarParser#cardDeclaration}.
         * @param ctx the parse tree
         */

```

```
void
enterCardDeclaration(CardStateDslGrammarParser.CardDeclarationContext ctx);
/**
 * Exit a parse tree produced by { @link
CardStateDslGrammarParser#cardDeclaration }.
 * @param ctx the parse tree
 */
void
exitCardDeclaration(CardStateDslGrammarParser.CardDeclarationContext ctx);
/**
 * Enter a parse tree produced by { @link
CardStateDslGrammarParser#stateDeclaration }.
 * @param ctx the parse tree
 */
void
enterStateDeclaration(CardStateDslGrammarParser.StateDeclarationContext ctx);
/**
 * Exit a parse tree produced by { @link
CardStateDslGrammarParser#stateDeclaration }.
 * @param ctx the parse tree
 */
void
exitStateDeclaration(CardStateDslGrammarParser.StateDeclarationContext ctx);
/**
 * Enter a parse tree produced by { @link
CardStateDslGrammarParser#transitionDeclaration }.
 * @param ctx the parse tree
 */
```

```

void
enterTransitionDeclaration(CardStateDslGrammarParser.TransitionDeclarationConte
xt ctx);
    /**
     * Exit a parse tree produced by { @link
CardStateDslGrammarParser#transitionDeclaration }.
     * @param ctx the parse tree
     */
void
exitTransitionDeclaration(CardStateDslGrammarParser.TransitionDeclarationContex
t ctx);
    /**
     * Enter a parse tree produced by { @link
CardStateDslGrammarParser#expression }.
     * @param ctx the parse tree
     */
void enterExpression(CardStateDslGrammarParser.ExpressionContext
ctx);
    /**
     * Exit a parse tree produced by { @link
CardStateDslGrammarParser#expression }.
     * @param ctx the parse tree
     */
void exitExpression(CardStateDslGrammarParser.ExpressionContext
ctx);
}
// Generated from
/Users/mykhailo/Documents/dsl_demo/app/src/main/java/com/mkhtk/dsl_demo/antlr/
CardStateDslGrammar.g4 by ANTLR 4.13.1
package com.mkhtk.antlr;

```

```

import org.antlr.v4.runtime.Lexer;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.atn.*;
import org.antlr.v4.runtime.dfa.DFA;
import org.antlr.v4.runtime.misc.*;

    @SuppressWarnings({"all", "warnings", "unchecked", "unused", "cast",
"CheckReturnValue", "this-escape"})
    public class CardStateDslGrammarLexer extends Lexer {
        static { RuntimeMetaData.checkVersion("4.13.1",
RuntimeMetaData.VERSION); }

        protected static final DFA[] _decisionToDFA;
        protected static final PredictionContextCache _sharedContextCache =
            new PredictionContextCache();
        public static final int
            T__0=1, T__1=2, T__2=3, T__3=4, T__4=5, T__5=6, T__6=7,
T__7=8, T__8=9,
            T__9=10, T__10=11, T__11=12, ID=13, INT=14, WS=15;
        public static String[] channelNames = {
            "DEFAULT_TOKEN_CHANNEL", "HIDDEN"
        };

        public static String[] modeNames = {
            "DEFAULT_MODE"
        };

```

```

private static String[] makeRuleNames() {
    return new String[] {
        "T__0", "T__1", "T__2", "T__3", "T__4", "T__5", "T__6",
"T__7", "T__8",
        "T__9", "T__10", "T__11", "ID", "INT", "WS"
    };
}
public static final String[] ruleNames = makeRuleNames();

private static String[] makeLiteralNames() {
    return new String[] {
        null, "Card", "Priority", "{", "}", "State", "on",
"become",
        ";", "if", "(", ")", "is"
    };
}
private static final String[] _LITERAL_NAMES = makeLiteralNames();
private static String[] makeSymbolicNames() {
    return new String[] {
        null, null, null, null, null, null, null, null, null, null,
null,
        null, "ID", "INT", "WS"
    };
}
private static final String[] _SYMBOLIC_NAMES =
makeSymbolicNames();
public static final Vocabulary VOCABULARY = new
VocabularyImpl(_LITERAL_NAMES, _SYMBOLIC_NAMES);

/**

```

```

* @deprecated Use { @link #VOCABULARY } instead.
*/
@Deprecated
public static final String[] tokenNames;
static {
    tokenNames = new String[_SYMBOLIC_NAMES.length];
    for (int i = 0; i < tokenNames.length; i++) {
        tokenNames[i] = VOCABULARY.getLiteralName(i);
        if (tokenNames[i] == null) {
            tokenNames[i] =
VOCABULARY.getSymbolicName(i);
        }

        if (tokenNames[i] == null) {
            tokenNames[i] = "<INVALID>";
        }
    }
}

@Override
@Deprecated
public String[] getTokenNames() {
    return tokenNames;
}

@Override

public Vocabulary getVocabulary() {
    return VOCABULARY;
}

```

```

public CardStateDslGrammarLexer(CharStream input) {
    super(input);
    _interp = new
LexerATNSimulator(this,_ATN,_decisionToDFA,_sharedContextCache);
}

@Override
public String getGrammarFileName() { return
"CardStateDslGrammar.g4"; }

@Override
public String[] getRuleNames() { return ruleNames; }

@Override
public String getSerializedATN() { return _serializedATN; }

@Override
public String[] getChannelNames() { return channelNames; }

@Override
public String[] getModeNames() { return modeNames; }

@Override
public ATN getATN() { return _ATN; }

public static final String _serializedATN = constantList

public static final ATN _ATN =

```

```
new
ATNDeserializer().deserialize(_serializedATN.toCharArray());
    static {
        _decisionToDFA = new DFA[_ATN.getNumberOfDecisions()];
        for (int i = 0; i < _ATN.getNumberOfDecisions(); i++) {
            _decisionToDFA[i] = new DFA(_ATN.getDecisionState(i),
i);
        }
    }
}
```