

Методичні вказівки
до виконання самостійної роботи з дисципліни
«Проектування додатків для мобільних пристроїв»
зі спеціальності «Комп'ютерна інженерія»

Міністерство освіти і науки України
Вінницький національний технічний університет

**Методичні вказівки
до виконання самостійної роботи з дисципліни
«Проектування додатків для мобільних пристроїв»
зі спеціальності «Комп'ютерна інженерія»**

Вінниця
ВНТУ
2026

Рекомендовано до видання Радою з якості освіти Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 13 від 18.06.2026 р.)

Рецензенти:

І. В. Богач, кандидат технічних наук, доцент

Н. П. Бабюк, кандидат технічних наук, доцент

Методичні вказівки до виконання самостійної роботи з дисципліни «Проектування додатків для мобільних пристроїв» зі спеціальності «Комп'ютерна інженерія» / уклад.: О. Г. Муращенко, О. С. Городецька. Електрон. текст. дані. Вінниця : ВНТУ, 2026. 43 с.

Методичні вказівки призначені для студентів денної та заочної форм навчання спеціальності «Комп'ютерна інженерія», які вивчають курс «Проектування додатків для мобільних пристроїв». Методичні вказівки містять теоретичний матеріал для самостійних робіт студентів, також наведено комплект тестових завдань, які студент може використовувати для самоперевірки.

ЗМІСТ

ВСТУП.....	4
1 ЗМІСТ НАВЧАЛЬНОЇ ДИСЦИПЛІНИ	5
2 ЗАГАЛЬНІ ВКАЗІВКИ ЩОДО ОРГАНІЗАЦІЇ САМОСТІЙНОЇ РОБОТИ	8
3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ	10
3.1 Узагальнена архітектура мобільного пристрою. Архітектура iOS та Android. Життєві цикли iOS та Android додатків ..	10
3.2 Сучасні підходи до створення додатків для мобільних пристроїв.....	13
3.3 Інструменти інтегрованого середовища програмування xCode	17
3.4 Інструменти інтегрованого середовища програмування Android Studio.....	19
3.5 Побудова користувацького інтерфейсу в iOS засобами SwiftUI.....	21
3.6 Побудова користувацького інтерфейсу в Android засобами Jetpack Compose.....	25
3.7 Сучасні архітектурні принципи та підходи до побудови додатків для мобільних пристроїв	28
4 ПЕРЕЛІК ПИТАНЬ ДЛЯ САМОКОНТРОЛЮ	32
5. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ	35
РЕКОМЕНДОВАНА ЛІТЕРАТУРА	41

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким поширенням мобільних пристроїв та зростанням ролі мобільних додатків у різних сферах діяльності людини – від комунікації та освіти до бізнесу, медицини та розваг. Смартфони, планшети та інші портативні пристрої стали невід’ємною частиною цифрової інфраструктури, а мобільні додатки – ключовим інструментом взаємодії користувача з інформаційними системами. У зв’язку з цим підготовка фахівців у галузі комп’ютерної інженерії потребує ґрунтовного опанування принципів, методів і засобів проєктування програмного забезпечення для мобільних платформ.

Актуальність вивчення дисципліни «Проєктування додатків для мобільних пристроїв» зумовлена високим попитом на кваліфікованих розробників мобільних застосунків, постійним оновленням технологій (зокрема iOS та Android), а також необхідністю створення ефективних, безпечних, масштабованих та зручних у використанні програмних продуктів. Особливого значення набувають сучасні підходи до розробки інтерфейсів користувача, архітектурні патерни, оптимізація продуктивності та забезпечення інформаційної безпеки мобільних додатків.

Метою методичних вказівок є надання студентам систематизованих рекомендацій щодо організації та виконання самостійної роботи з дисципліни, поглиблення теоретичних знань і формування практичних навичок проєктування, розробки та аналізу мобільних додатків для платформ iOS та Android.

Основними завданнями вивчення дисципліни та самостійної роботи студентів є: засвоєння базових понять і класифікацій мобільних пристроїв та додатків; вивчення архітектури мобільних операційних систем та життєвих циклів мобільних додатків; опанування сучасних інструментів розробки, зокрема Xcode та Android Studio; формування навичок побудови користувацьких інтерфейсів із використанням SwiftUI та Jetpack Compose; дослідження сучасних архітектурних підходів (MVC, MVVM, MVP) у мобільній розробці; набуття практичних умінь створення, тестування та оптимізації мобільних додатків; розвиток навичок самостійного аналізу, узагальнення та застосування отриманих знань при вирішенні прикладних завдань.

Методичні вказівки структуровано відповідно до змісту навчальної дисципліни та містять рекомендації щодо організації самостійної роботи, завдання для індивідуального опрацювання, питання для самоконтролю та тестові завдання. Їх використання сприятиме підвищенню рівня професійної підготовки студентів, розвитку аналітичного мислення та формуванню компетентностей, необхідних для успішної діяльності у сфері розробки мобільного програмного забезпечення.

1 ЗМІСТ НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Змістовий модуль 1. Мобільні пристрої. Додатки для мобільних ОС Android та iOS.

Тема 1. Класифікація мобільних пристроїв. Класифікація додатків для мобільних пристроїв

Основні типи мобільних пристроїв (смартфони, планшети, розумні годинники тощо). Характеристики апаратного забезпечення мобільних пристроїв. Класифікація мобільних додатків: нативні, гібридні, веб-додатки. Переваги та недоліки різних типів мобільних додатків. Вибір платформи для розробки: iOS, Android, кросплатформні рішення.

Тема 2. Загальна структура та принципи побудови мобільних операційних систем

Основні компоненти мобільних ОС (ядро, драйвери, API, системні сервіси). Взаємодія мобільної ОС з апаратним забезпеченням. Модель безпеки та управління правами доступу в мобільних ОС. Життєвий цикл додатків у мобільних ОС. Підхід "пісочниці" (sandboxing) у мобільних ОС.

Тема 3. Принципи побудови програмного забезпечення для мобільних пристроїв

Архітектура мобільних додатків (MVC, MVVM, MVP). Оптимізація продуктивності мобільних додатків. Використання патернів проектування у мобільній розробці. Зберігання та обробка даних у мобільних додатках. Безпека мобільних додатків: шифрування, автентифікація, захист даних.

Тема 4. Сучасні мобільні операційні системи iOS та Android

Архітектура Android та iOS: схожості та відмінності. Магазины додатків: Google Play vs. App Store. Основні мови програмування для Android (Java/Kotlin) та iOS (Swift). Політика безпеки та конфіденційності в Android та iOS. Особливості розповсюдження та оновлення додатків на різних платформах.

Тема 5. Створення простого додатку для мобільних пристроїв на базі ОС Android

Налаштування середовища розробки (Android Studio, SDK). Основи мови Kotlin або Java для Android. Структура проекту Android-дodatка. Життєвий цикл активностей (Activity). Робота з ресурсами та макетами (Layouts).

Тема 6. Створення простого додатку для мобільних пристроїв на базі ОС iOS

Встановлення та налаштування Xcode та iOS SDK. Основи Swift для розробки iOS-додатків. Структура проекту iOS-дodatка. Життєвий цикл контролерів вигляду (ViewControllers). Автоматичне компоновання інтерфейсу (Auto Layout).

Тема 7. Порівняльний аналіз структури та вмісту додатків для iOS та Android

Основні архітектурні відмінності iOS та Android-додатків. Порівняння життєвого циклу додатків на Android та iOS. UI/UX відмінності між iOS та Android (Material Design vs. Human Interface Guidelines). Підходи до роботи з мережею та базами даних. Монетизація додатків: Google Play Billing vs. App Store In-App Purchases.

Змістовий модуль 2. Розробка програмного забезпечення для мобільних пристроїв.

Тема 8. Touch UI: Основи розробки інтерфейсів користувача

Основи побудови адаптивних UI для мобільних пристроїв. Принципи Mobile-first дизайну. Використання кольорів, типографіки та іконографії. Респонсивність інтерфейсу: портретний та ландшафтний режими. Доступність (accessibility) у мобільних додатках.

Тема 9. Touch UI: Основи розробки багатоекранних та багатовіконних додатків

Робота з навігацією між екранами. Навігаційні компоненти: Bottom Navigation, Drawer, Tab Bar. Використання фрагментів в Android та сторібордів (Storyboards) в iOS. Управління стеком екранів та перехід між активностями. Оптимізація продуктивності UI для плавної навігації.

Тема 10. Touch UI: Елементи керування вибором. Таблиці (TableView). Прокрутка та масштабування

Використання списків та таблиць: RecyclerView (Android) та UITableView (iOS). Робота з адаптерами та джерелами даних у списках. Реалізація прокрутки та масштабування в мобільних UI. Ліниве завантаження даних (lazy loading).

Тема 11. Touch UI: Контролери вигляду (ViewControllers) та Активності (Activities)

Основи роботи з контролерами вигляду в iOS. Основи роботи з активностями та фрагментами в Android. Використання контейнерних контролерів та навігаційних компонентів. Управління станом контролерів вигляду та активностей.

Тема 12. Touch UI: Обробка дотиків та жестів

Основні типи жестів: дотик, свайп, масштабування, довгий тап. Використання Gesture Recognizers в iOS та GestureDetector в Android. Реалізація мультитач-взаємодії. Робота з Drag & Drop.

Тема 13. Динамічне оновлення екрану. Анімація та 2D графіка

Використання анімацій у мобільних додатках. Бібліотеки анімацій: Lottie, MotionLayout. Оптимізація продуктивності анімацій. Основи роботи з Canvas API та 2D графікою.

**Тема 14. Позиціонування в просторі. Сенсори та Geolocation.
Робота з картами**

Використання GPS, акселерометра, гіроскопа. Робота з картографічними сервісами (Google Maps API, MapKit). Використання геолокаційних запитів та відстеження місцезнаходження. Оптимізація використання батареї при роботі з сенсорами.

Тема 15. Робота з мультимедіа. Аудіо, фото, відео

Відтворення та запис аудіо та відео. Використання Media Player API та AVFoundation. Робота з камерою та галереєю (Camera API, Image Picker). Обробка зображень та відео. Стрімінгові сервіси та обробка мультимедійного контенту в реальному часі.

2 ЗАГАЛЬНІ ВКАЗІВКИ ЩОДО ОРГАНІЗАЦІЇ САМОСТІЙНОЇ РОБОТИ

Самостійна робота студентів в межах дисципліни «Проектування додатків для мобільних пристроїв» виконується відповідно до чинного Положення про організацію самостійної роботи здобувачів вищої освіти ВНТУ (<https://vntu.edu.ua/uploads/n/np/4.pdf>) [1]. Самостійна робота спрямована на закріплення теоретичних знань, які студент отримує під час навчання, їх поглиблення, а також набуття та удосконалення практичних навичок із проектування додатків для мобільних ОС Android та iOS.

Самостійна робота виконується студентом під методичним керівництвом викладача, але без його безпосередньої участі. Самостійна робота з дисципліни «Проектування додатків для мобільних пристроїв» включає такі види:

- робота над окремими темами теоретичного матеріалу, які винесені на самостійне опрацювання студентами і будуть наведені нижче;
- опрацювання лекційного матеріалу;
- підготовка до практичних занять;
- підготовка до контрольних випробувань, а саме до модульного та підсумкового контролю.

Основні види самостійної роботи студента із витратами часу наведені в таблиці 1.

Таблиця 1 – Самостійна робота студентів

№ з/п	Назва теми	Кількість годин
1	Самостійне опрацювання тем теоретичного матеріалу	55
2	Опрацювання лекційного матеріалу (36 годин)	10
3	Підготовка до практичних занять (13 занять)	13
4	Підготовка до модульного контролю (два колоквиуми)	4
5	Підготовка до складання диф. заліку	5
	Усього	87

Робота над окремими темами теоретичного матеріалу, які винесені на самостійне опрацювання студентами. Цей вид діяльності передбачає глибоке індивідуальне вивчення студентами визначених тем, що не розглядаються детально під час лекцій. Метою є формування повного уявлення про концепції, технології та інструменти, що використовуються у процесі проектування мобільних додатків. Самостійне опрацювання передбачає роботу з навчальними посібниками, науковими публікаціями, онлайн-курсами та

ресурсами, документацією до програмних засобів тощо. Підсумком цієї роботи можуть бути короткі реферати, конспекти або презентації.

Опрацювання лекційного матеріалу. Цей вид роботи передбачає повторне вивчення та осмислення інформації, отриманої на лекціях. Студент повинен самостійно переглянути конспекти, розібратися у складних моментах, доповнити матеріал додатковими джерелами. Це сприяє закріпленню знань, систематизації інформації та кращій підготовці до виконання практичних завдань і контрольних випробувань. Також рекомендується складання опорних схем або карт знань з ключових тем.

Підготовка до практичних занять. Підготовка до практичних занять включає ознайомлення з теоретичними основами майбутньої роботи, встановлення необхідного програмного забезпечення, налаштування середовищ розробки, а також виконання попередніх завдань, якщо вони передбачені. Мета – забезпечити студенту можливість ефективно працювати на практичному занятті, фокусуючись безпосередньо на реалізації завдань, а не на вирішенні організаційних або підготовчих питань.

Підготовка до контрольних випробувань, а саме до модульного та підсумкового контролю. Цей вид самостійної роботи має на меті повторення, узагальнення та систематизацію знань, отриманих протягом семестру. Студенти повинні опрацювати список контрольних питань, відпрацювати навички, пов'язані з написанням коду, аналізом архітектури мобільних додатків, а також перевірити свої знання за допомогою тестів або прикладів завдань з попередніх модулів. Ретельна підготовка до контрольних випробувань сприяє успішному засвоєнню матеріалу дисципліни.

3 ЗАВДАННЯ ДЛЯ САМОСТІЙНОЇ РОБОТИ СТУДЕНТІВ

3.1 Узагальнена архітектура мобільного пристрою. Архітектура iOS та Android. Життєві цикли iOS та Android додатків

Мета: вивчення структури та складових компонентів і частин апаратного та програмного забезпечення мобільних пристроїв.

Сучасний мобільний пристрій – це складна система, де апаратне забезпечення тісно взаємодіє з багаторівневим програмним середовищем. **Апаратна частина** надає обчислювальні можливості, забезпечує роботу фізичних компонентів системи, тоді як **програмна частина** реалізує керування пристроєм, взаємодію з апаратним забезпеченням, бізнес-логіку, графічний інтерфейс та підтримку взаємодії з користувачем. Розуміння архітектури мобільного пристрою є фундаментальним для ефективної розробки додатків для мобільних пристроїв, зокрема пристроїв з операційними системами Android та iOS.

Узагальнену архітектуру мобільного пристрою наведено на рис. 1.

Апаратна частина сучасного мобільного пристрою (смартфона або планшета) – це компактна, інтегрована система, створена для забезпечення функціональності, продуктивності, енергоефективності і компактності системи.

Основними **апаратними компонентами** сучасного мобільного пристрою є:

- Центральний процесор (ЦП або CPU) – головний обчислювальний блок, який виконує інструкції операційної системи та додатків. В мобільних пристроях часто використовуються енергоефективні ARM-процесори (наприклад, Qualcomm Snapdragon, Apple A-series, MediaTek та інші).
- Графічний процесор (ГП або GPU) – обробляє графіку, анімацію, відео та візуальні елементи інтерфейсу. Відповідає за продуктивність у відеоіграх, візуалізацію інтерфейсу та AR/VR-додатки.
- Оперативна пам'ять (ОЗП або RAM) – тимчасове сховище для даних, які використовуються активними додатками. Впливає на швидкість перемикання між задачами й загальну продуктивність.
- Постійна пам'ять (ПЗП або Flash Storage) – зберігає ОС, додатки, мультимедійні файли та інші дані. Підключена як eMMC або UFS, що впливає на швидкодію.

Мобільний Пристрій



Рисунок 1 – Узагальнена архітектура мобільних пристроїв

- Сенсорний дисплей (Touch Screen) – поєднує функції екрана та сенсора. Реалізує інтерфейс взаємодії з користувачем. Зазвичай використовуються OLED або IPS-матриці.
- Модулі зв'язку:
 - Wi-Fi / Bluetooth / NFC – слугує для передачі даних на коротких відстанях.
 - Мобільний модем (LTE/5G) – забезпечує зв'язок з мобільними мережами.
 - GPS – надає дані глобального позиціонування.

- Сенсори:
 - Акселерометр, гіроскоп, цифровий компас або магнітометр – призначені для визначення положення й руху пристрою.
 - Датчик наближення, освітлення – призначені для визначення і контролю яскравості екрана, автоматичного вимкнення і т. ін.
 - Біометричні сенсори – сканери відбитків пальців або сітківки ока, Face ID.
- Система живлення та акумулятор – джерело живлення пристрою. Складовою частиною є сучасні акумуляторні батареї, які базуються на літієво-іонній або літієво-полімерній технології, часто з підтримкою швидкої або бездротової зарядки.

Програмна частина мобільного пристрою формує програмний стек, який забезпечує керування апаратними ресурсами, відповідає за безперервну роботу пристрою та запуск і роботу системних і користувацьких додатків. В загальному, можна виділити наступні основні **програмні компоненти**:

- Ядро операційної системи (Kernel Layer) – проганий шар, що забезпечує низькорівневий доступ до апаратного забезпечення, є основною платформою для запуску програм, транслює програмні інструкції та забезпечує керування пам'яттю, процесами, файлами, драйверами пристроїв, мережею, безпекою і т.ін.. В ОС Android ядро операційної системи створена компанією Google, засноване на Linux, має відкритий вихідний код і доступне для зміни і кастомізації різними виробниками пристроїв. Підтримує мови програмування Java, Kotlin та C++. На відміну від Android, ядро операційної системи iOS є власницьким продуктом компанії Apple і засноване на XNU ("X is Not Unix") і закрите для будь-яких сторонніх змін поза контролем компанії Apple. Підтримує мови програмування Swift та Objective-C.
- Апаратна абстракція (HAL або Hardware Abstraction Layer) – програмний шар низького рівня, що забезпечує абстрактний і уніфікований інтерфейс між ядром і драйверами. HAL дозволяє ОС використовувати модулі (наприклад, камеру або Bluetooth) без знання деталей та конкретних особливостей їх реалізації.
- Системні бібліотеки та сервіси (Native Libraries & Runtime) – зокрема віртуальна машина для виконання додатків (Android Runtime (ART)), основні фреймворки для роботи з базовою функціональністю пристрою, зокрема з графікою, даними, touch-інтерфейсом, а також інші базові інтеграційні бібліотеки, наприклад SQLite, WebKit, OpenGL тощо.
- Сервіси та фреймворк прикладного рівня (Application Framework) – програмна надбудова вищого рівня абстракції для створення користувацьких додатків, яка надає API для доступу до сенсорів, елементів графічного інтерфейсу, мультимедіа, мережі тощо.

- Користувацькі додатки (Applications) – завершальний рівень, на якому користувач взаємодіє з пристроєм. Складається з системних додатків (навігатор, телефон, камера) та сторонніх програм, які користувач встановлює самостійно.

Під час підготовки до цієї теми необхідно самостійно розглянути питання:

- Архітектура iOS ОС. Core Services, Cocoa Touch, Media. Життєвий цикл додатку для операційної системи iOS.
- Архітектура Android ОС. Android Runtime (ART). Життєвий цикл додатку для операційної системи Android.
- Узагальнений життєвий цикл додатків для мобільних пристроїв. Порівняльна характеристика життєвих циклів додатку для операційних систем iOS та Android.

3.2 Сучасні підходи до створення додатків для мобільних пристроїв

Мета: огляд сучасних засобів та підходів до розробки додатків для мобільних пристроїв.

Проектування та розробка додатків для мобільних пристроїв – це процес створення програмного забезпечення, призначеного для роботи на мобільних пристроях, таких як смартфони, планшети, смарт-годинники тощо. Зі зростанням кількості мобільних користувачів у світі мобільні додатки стали критично важливими для бізнесу, освіти, розваг, медицини та інших сфер. У теперішній час у світі домінують дві мобільні операційні системи: iOS від компанії Apple та Android, що розроблена компанією Google, тож в рамках дисципліни «Проектування додатків для мобільних пристроїв» основний акцент зроблений саме на розробці додатків для операційних систем iOS та Android.

У таблиці 2 наведено деякі характерні особливості мобільних операційних систем iOS та Android, зокрема основні мови програмування, що застосовуються для створення програм.

Таблиця 2 – Порівняльна характеристика особливостей створення та роботи мобільних додатків для операційних систем Android та iOS

Характеристика	OC Android	OC iOS
Мови програмування (основні)	Kotlin, Java	Swift, Objective-C
Основне середовище розробки (IDE)	Android Studio	Xcode
Спосіб поширення додатків (магазин застосунків)	Google Play та інші	App Store
Можливість встановлювати сторонні додатки	Так	Ні
Тип системи	Система з відкритим кодом	Закрита система
Пристрої	Різні виробники	Лише Apple
Гнучкість UI	Висока	Обмежена Apple-гайдами
Інструменти для створення інтерфейсу користувача	XML Layouts, Jetpack Compose	UIKit, SwiftUI
Монетизація	Гнучкі можливості	Строгий контроль Apple
Оновлення системи	Повільні, залежать від виробника	Швидкі й одночасні

Варто зазначити, що обидві платформи та операційні системи постійно розвиваються і вдосконалюються, що призводить до змін у доступних підходах і інструментах розробників програмного забезпечення. Зокрема, щодо мов програмування, то у випадку iOS доцільно виділити:

- Objective-C – мова, яка довгий час була основною для iOS, але на сьогодні використовується переважно в проєктах, що знаходяться у фазі підтримки;

- Swift: сучасна мова програмування від Apple, яка прийшла на зміну Objective-C. Має лаконічний синтаксис, високу безпечність та хорошу продуктивність. В основі Swift лежать ідеї з безпечного управління пам'яттю, типової безпеки (статичної типізації) та підтримки декларативного стилю коду.

До основних мов програмування для OC Android варто віднести:

- Java: традиційна мова для розробки під Android. Вона забезпечує стабільність і підтримку великої кількості бібліотек, однак має відносно застарілий синтаксис;

- Kotlin: рекомендована мова від Google з 2017 року. Має сучасний синтаксис, підтримку функціонального програмування, розширення для Android SDK, сумісність з Java. Kotlin активно використовується у нових проєктах, особливо у зв'язці з Jetpack Compose.

Щодо підходів до розробки інтерфейсів користувача (UI), то тут також варто відзначити певні особливості кожної платформи. Зокрема, в Android існує наступних два підходи:

- використання XML Layouts – традиційний підхід до опису графічного інтерфейсу користувача, в якому кожен елемент UI визначається у спеціальних XML-файлах, а логіка реалізується у Kotlin або Java;

- використання Jetpack Compose – сучасний декларативний підхід, що дозволяє створювати UI безпосередньо у коді за допомогою функцій-компонентів. Перевагами цього є спрощена робота з UI, менше шаблонного коду, що дублюється, інтеграцію зі станами елементів та ін.

Водночас, у випадку розробки програмних продуктів для операційної системи iOS існують такі підходи і відповідні їм інструменти:

- використання UIKit – імперативний фреймворк для створення інтерфейсу користувача. Інтерфейси створюються розробником в програмному коді або за допомогою Interface Builder у Xcode;

- використання SwiftUI – декларативний фреймворк, представлений у 2019 році. Дозволяє будувати UI з використанням синтаксису Swift, з фокусом на автоматичному оновленні інтерфейсу у відповідь на зміни стану додатку.

Вказані вище особливості є лише деякими факторами, що впливають на остаточний вибір технологій, інструментів та підходів під час розробки програмного забезпечення для мобільних пристроїв. Варто зазначити, що в комерційній та промисловій сфері досить поширеною є ситуація, коли програмний продукт має бути присутнім на обох згаданих платформах, тобто постає питання підтримки так званої мультиплатформеності. Щодо цього питання, на сьогодні доцільно виділити три основні підходи до розробки мобільних додатків для різних ОС, а саме:

1. Нативна розробка (native development) – розробка окремих незалежних додатків для кожної платформи з використанням офіційних рекомендованих мов та інструментів. Перевагами такого підходу є висока продуктивність, доступ до всіх API платформи, краща інтеграція з апаратним забезпеченням, а також відносна доступність офіційної документації, довідкових матеріалів, порад і найкращих практик від виробників ОС. Основним недоліком можна вважати необхідність створення і підтримки двох окремих проєктів (Android + iOS), а отже і двох інженерних команд. Цей підхід є найбільш гнучким і не має додаткових обмежень, проте може бути більш вартісним і вимагати більше витраченого часу на розробку, оскільки щоразу потрібно реалізовувати нову функціональність чи вносити зміни до існуючої окремо для кожної платформи.

2. Кросплатформна розробка – передбачає використання єдиної бази коду для обох платформ за допомогою спеціальних інструментів і засобів. Найбільш популярними серед таких засобів є:

- Flutter – кросплатформний програмний фреймворк, розроблений компанією Google і призначений для створення iOS, Android та Web додатків. Мова програмування – Dart.

- React Native – програмний фреймворк з відкритим кодом від компанії Meta Platforms (Facebook), призначений для створення додатків для iOS, MacOS, tvOS, Android, Android TV, та Windows та Web. Використовує підходи React та JavaScript/TypeScript.

Найсуттєвішими перевагами цього підходу є наявність спільної бізнес-логіки програм для різних платформ, що, в свою чергу, призводить до зменшення часу і вартості розробки. Серед потенційних недоліків можна відмітити можливу нижчу продуктивність/швидкість додатку, обмеження доступу до специфічних API платформ, відсутність підтримки виробників операційних систем, тощо.

3. Гібридна розробка – підхід, при якому застосунок створюється як вебдодаток (англійською мовою іноді використовують термін web-wrapper), що запускається всередині вебконтейнера (WebView), тобто як вебзастосунок. В якості прикладів таких інструментів можна навести фреймворки Ionic, Apache Cordova та деякі інші. Суттєвими перевагами цього підходу є швидкий старт розробки (не вимагає специфічних знань платформи від розробників) та можливість використання вебстеків (HTML/CSS/JS). Мінусами є слабка продуктивність додатків та обмежена взаємодія з апаратним забезпеченням. Цей підхід доцільно використовувати для швидкого створення нескладних додатків, які не вимагають глибокої інтеграції з апаратними засобами та специфічними платформозалежними API, або у випадку швидкого портування існуючого програмного коду з веб на інші платформи.

Таким чином, сучасна розробка програмного забезпечення для мобільних пристроїв є багатогранною галуззю, що охоплює різні підходи, мови програмування, фреймворки та інструменти. Вибір технологій та архітектурних рішень залежить від специфіки проєкту, наявних ресурсів та цільової аудиторії, особливостей необхідного користувацького досвіду, дизайну та підтримки продукту.

Під час підготовки до цієї теми необхідно самостійно розглянути питання:

- Порівняльна характеристика нативного (native development) та кросплатформного (cross-platform development) підходів до розробки додатків для мобільних пристроїв.
- Спільні та відмінні риси платформ Android та iOS.

3.3 Інструменти інтегрованого середовища програмування xCode

Мета: дослідження структури інтегрованого середовища програмування xCode для розробки мобільних додатків для операційної системи iOS.

Xcode – це офіційне інтегроване середовище програмування (англ. IDE – Integrated development environment) виробництва компанії Apple, що призначене для розробки програмного забезпечення для сімейств операційних систем iOS (iPhone, iPad), macOS, watchOS та tvOS. Середовище **Xcode** підтримує основні мови програмування **Swift** та **Objective-C**, має потужний візуальний редактор інтерфейсу та інструменти для тестування, дебагінгу й публікації додатків. На рис. 2 наведено ілюстрацію головного вікна xCode IDE, яке складається з наступних частин:

– Навігаційна панель (Navigator area) – відповідає за відображення, навігацію і модифікацію структури проєкту та складається з наступних основних частин:

- Project Navigator – відповідає за роботу з деревом файлів;
- Symbol Navigator – відповідає за роботу з класами, методами, властивостями і т.ін.;
- Issue Navigator – відповідає за роботу з помилками компіляції, попередженнями та іншою інформацією щодо можливих недоліків чи вразливостей в програмному коді;
- Test / Debug / Breakpoint Navigator – відповідає за роботу з інструментами відлагодження, відображення поточного значення змінних, точок зупинки додатку та контролю його поточного стану.

– Панель редагування (Editor area) – частина візуального інтерфейсу, що забезпечує:

- редагування програмного коду;
- редагування візуальних елементів в режимі Interface Builder;
- підтримку підказок, автозаповнення і т.ін.;
- можливість відображення на екрані програмного коду та візуального дизайну одночасно.

– Панель налаштувань (Inspector area), що відповідає за:

- параметри об'єктів (назви, властивості);
- атрибути, обмеження Auto Layout;
- зв'язування даних (Data Binding) для режиму роботи з SwiftUI.

– Панель інструментів (Toolbar), що включає в себе:

- кнопки запуску проєкту, вибору пристрою, стан останньої збірки та її дату і т.ін.;
- поточну гілку репозиторію git (якщо проєкт підключено до git).

– Панель відлагодження (Debug area), яка відповідає за:

- роботу з консоллю логів виконання;
- перегляд поточного значення змінних та об'єктів під час виконання програми.



Рисунок 2 – Головне вікно середовища програмування xCode
(з офіційної документації Apple, посилання:
<https://developer.apple.com/documentation/xcode/creating-an-xcode-project-for-an-app>)

Окрім усіх зазначених інструментів роботи з вихідним програмним кодом, Xcode також включає в себе більшу частину документації розробника, яку надає компанія Apple, а також окремий інструмент для створення графічних інтерфейсів – **Interface Builder (IB)**. Історично цей інструмент був навмисно розроблений як окрема програма, щоб дозволити дизайнерам розробляти графічні інтерфейси без використання орієнтованої на код IDE, але згодом Apple інтегрувала його функціональні можливості безпосередньо в Xcode. Interface Builder надає розробнику колекції об'єктів інтерфейсу користувача. Ці об'єкти інтерфейсу користувача містять такі елементи, як текстові поля, таблиці даних, повзунки та спливаючі меню і т.ін. Усі зазначені елементи повністю розширювані, тобто будь-який розробник може розробляти нові об'єкти, модифікувати існуючі та додавати їх до колекцій IB. Отриманий інтерфейс зберігається у вигляді файлів спеціального формату .xib на основі XML.

Під час підготовки до цієї теми необхідно самостійно розглянути питання:

- Створення візуальної частини додатків засобами інструменту Interface Builder, інтерактивне відображення та синхронізація програмного коду з графічним інтерфейсом .xib-файлів в xCode.

- Робота з ресурсними файлами проєкту: графічні файли, файли з текстовими рядками і локалізацією, файли з форматованими даними в xCode.
- Налаштування та відлагодження проєкту (англ. tuning and debugging): робота з реальними мобільними пристроями та їх симуляторами, інструменти тестування та відлагодження, вимірювання ефективності та збирання метрик.

3.4 Інструменти інтегрованого середовища програмування Android Studio

Мета: дослідження структури інтегрованого середовища програмування Android Studio для розробки мобільних додатків для операційної системи Android.

Android Studio – це офіційне інтегроване середовище програмування від компанії Google, призначене для створення додатків для сімейства мобільних операційних системи Android. Android Studio побудована на базі IDE IntelliJ IDEA та підтримує розробку на мовах програмування Kotlin, Java та C++ і має повну інтеграцію з Android SDK, емулятором, Gradle та іншими інструментами розробки, відлагодження і публікації програмних продуктів. На рис. 3 наведено ілюстрацію головного вікна Android Studio IDE, яке складається з наступних частин, позначених відповідними номерами:

1. Панель інструментів (Toolbar) – включає в себе різні потенційні дії роботи з проєктом, зокрема:

- збірка та старт виконання додатка (кнопки Run, Debug і т.ін.);
- вибір емулятора, на якому запускається додаток;
- швидкий доступ до різних Android-інструментів.

2. Навігаційна панель (Navigation bar) – відповідає за відображення дерева структури проєкту та надає змогу відкривати та редагувати файли.

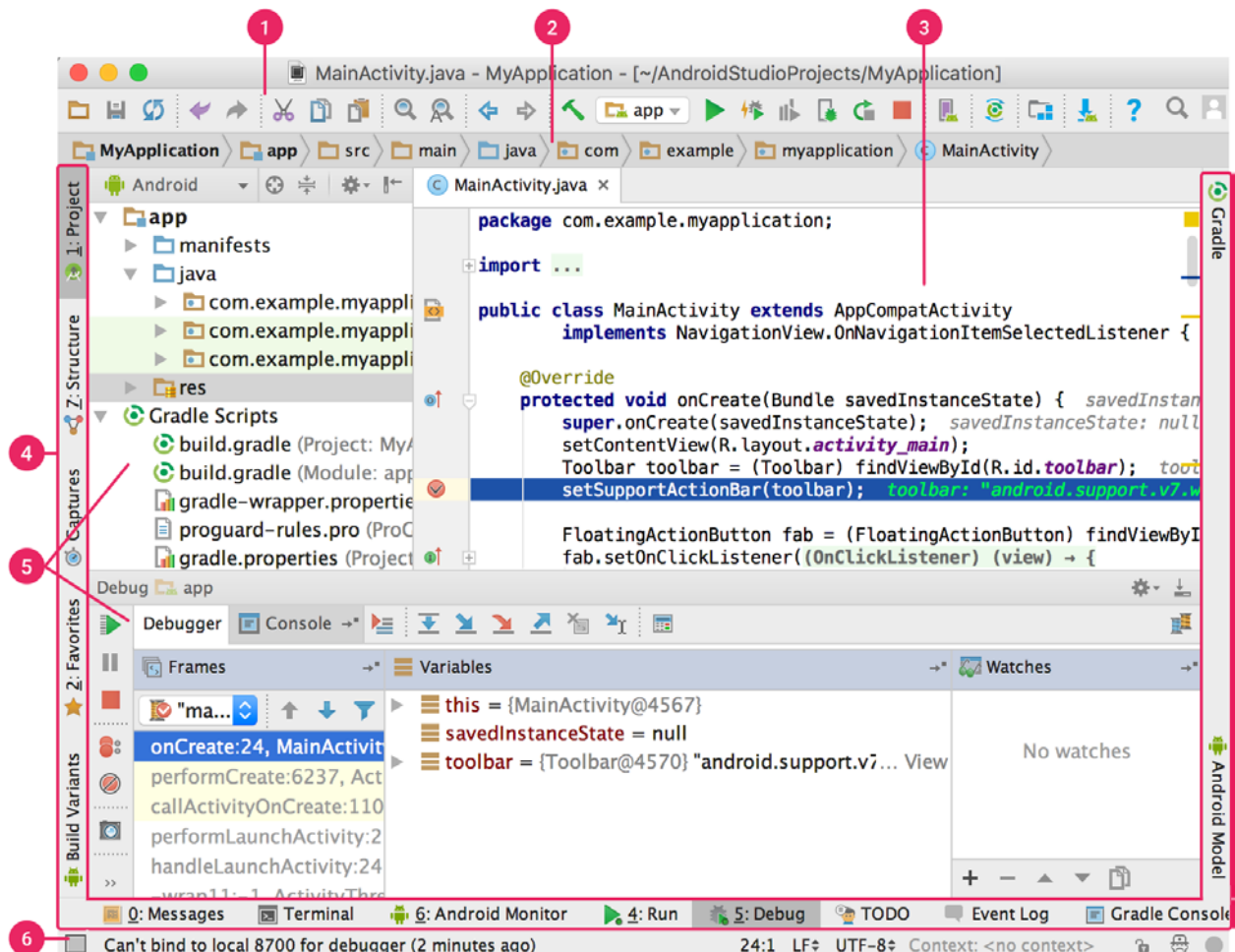


Рисунок 3 – Головне вікно середовища програмування Android Studio
(з офіційної документації Adnroid Developer Portal, посилання:
<https://developer.android.com/studio/intro/user-interface>)

3. Вікно редагування коду (Editor window) – основна частина IDE, що призначена для створення і модифікації програмного коду. Залежно від типу файлу, що вибрано для відображення, це вікно може набувати різного вигляду і надавати різні інструменти редагування. Зокрема, набуває зовсім іншого вигляду в режимі редагування графічної розмітки інтерфейсу Layout Editor.

4. Панель активних вікон інструментів (Tool window bar) – надає швидкий доступ до окремих інструментів IDE та розкриває або згортає вікно відповідного інструменту.

5. Активні вікна інструментів (Tool windows) – доступ та відображення вікон спеціальних інструментів та задач, наприклад:

- a. журнал подій додатку Logcat;
- b. панелі виконання (runtime) програмного коду, такі як Profiler, Terminal, Debugger, Console;
- c. інструменти пошуку, модифікації та збереження програмного коду (search panel, version control і т. ін.).

6. Панель поточного стану (Status bar) – відображає поточний стан збірки проєкту, помилки та попередження виконання, фонові задачі та інші повідомлення.

Варто зазначити, що як і у випадку Xcode і Interface Builder, в Android Studio також існує спеціальний інструмент для роботи з графічним інтерфейсом – **Layout Editor**. Цей інструмент призначений для створення та редагування інтерфейсу користувача (UI) додатку, що розробляється і його основною задачею є створення макета екрана та/або окремих елементів екрана у вигляді ієрархічної структури контейнерів та інших графічних компонентів, збережених в спеціальному форматі, основаному на XML. Layout Editor дозволяє розробнику зручно комбінувати візуальний підхід розробки, оснований на механізмі drag-and-drop, із текстовим редагуванням XML-файлів, та одразу перевіряти результат застосування змін. Насьогодні Layout Editor за замовчуванням використовує ConstraintLayout – гнучкий контейнер, який дозволяє "прикріплювати" елементи до інших елементів та/або меж контейнера замість задання «жорсткої» позиції на екрані шляхом встановлення визначених координат.

Під час підготовки до цієї теми необхідно самостійно розглянути питання:

- Створення візуальної частини додатків на базі XML та Jetpack Compose.
- Робота з ресурсними файлами проєкту: графічні файли, файли з текстовими рядками і локалізацією, файли з форматованими даними в Android Studio.
- Налаштування та відлагодження Android проєкту: робота з реальними мобільними пристроями та Android Virtual Device (AVD). Інструменти тестування та відлагодження.

3.5 Побудова користувацького інтерфейсу в iOS засобами SwiftUI

Мета: вивчення інструментів та засобів побудови користувацького інтерфейсу мобільних додатків для операційної системи iOS засобами фреймворку SwiftUI.

SwiftUI – це сучасний фреймворк компанії Apple для побудови користувацьких інтерфейсів, представлений у 2019 році. Він призначений для розробки інтерфейсів застосунків для всіх платформ Apple – iOS, iPadOS, macOS, watchOS та tvOS – на основі єдиного підходу та мови програмування Swift. SwiftUI поступово стає основним рекомендованим інструментом побудови UI, проте співіснує з більш раннім імперативним фреймворком UIKit, який залишається широко використовуваним, зокрема у наявних (legacy) проєктах.

Ключовою особливістю SwiftUI є декларативна парадигма побудови інтерфейсу. На відміну від імперативного підходу (характерного для UIKit),

де розробник покроково описує, як саме створити та змінити елементи інтерфейсу, у декларативному підході розробник описує, яким інтерфейс має бути для певного стану даних, а фреймворк самостійно визначає, які зміни потрібно внести у відображення. Це суттєво зменшує обсяг шаблонного коду та знижує ймовірність помилок, пов'язаних із ручним синхронізуванням стану та інтерфейсу.

В основі SwiftUI лежить кілька фундаментальних понять:

- **View** (представлення) – основний будівельний блок інтерфейсу. У SwiftUI будь-який візуальний елемент є структурою (struct), що відповідає протоколу View та має обов'язкову обчислювану властивість `body`, яка повертає опис вмісту цього елемента;

- композиція – складні інтерфейси будуються шляхом поєднання та вкладення простих View один в одного, а не через успадкування, що робить код модульним і придатним для повторного використання;

- єдине джерело істини (single source of truth) – стан інтерфейсу зберігається в одному визначеному місці, а всі залежні елементи автоматично оновлюються при його зміні;

- реактивність – при зміні стану SwiftUI автоматично й ефективно перемальовує лише ті частини інтерфейсу, які залежать від цих даних.

Побудова інтерфейсу засобами SwiftUI здійснюється шляхом комбінування готових компонентів та контейнерів компонування. До базових елементів інтерфейсу належать: `Text` (текстовий напис), `Image` (зображення), `Button` (кнопка), `TextField` (поле введення тексту), `Toggle` (перемикач), `Slider` (повзунок) та інші. Для розташування елементів використовуються контейнери компонування:

- **VStack** – вертикальне розташування елементів (один під одним);
- **HStack** – горизонтальне розташування (один поряд з одним);
- **ZStack** – накладання елементів один на одного за віссю глибини;
- **List** – прокручуваний список елементів;
- **LazyVStack**, **LazyHStack** та **Grid** – контейнери для ефективного відображення великих наборів даних із «лінивим» завантаженням.

Зовнішній вигляд і поведінку елементів налаштовують за допомогою модифікаторів (modifiers) – спеціальних методів, що повертають змінену копію View. Модифікатори застосовуються послідовно, утворюючи ланцюжок (наприклад, задання шрифту, кольору, відступів, рамок). Розробник також може створювати власні (custom) View, виділяючи повторювані фрагменти інтерфейсу в окремі структури, що підвищує читабельність і повторне використання коду.

Приклад простого екрана SwiftUI, що містить заголовок, лічильник і кнопку для його збільшення, наведено нижче.

```

struct ContentView: View {
    @State private var count = 0

    var body: some View {
        VStack(spacing: 20) {
            Text("Лічильник: \(count)")
                .font(.title)
                .foregroundColor(.blue)

            Button("Збільшити") {
                count += 1
            }
                .padding()
                .background(Color.green)
                .cornerRadius(8)
        }
        .padding()
    }
}

```

Важливою складовою SwiftUI є механізм управління станом та зв'язування даних (Data Binding). Оскільки інтерфейс автоматично реагує на зміни стану, фреймворк надає набір спеціальних обгортки властивостей (property wrappers), що визначають, як саме дані пов'язані з інтерфейсом:

- **@State** – для зберігання локального змінюваного стану всередині View; зміна такого значення автоматично спричиняє перемальовування інтерфейсу;
- **@Binding** – створює двостороннє посилання на стан, що належить іншому View, дозволяючи дочірньому елементу читати й змінювати спільні дані;
- **@ObservedObject** та **@StateObject** – для роботи зі складними об'єктами моделі, що відповідають протоколу ObservableObject і повідомляють інтерфейс про зміни своїх властивостей (позначених @Published);
- **@EnvironmentObject** – для доступу до спільних даних, які передаються через ієрархію View без явного передавання їх у кожен елемент.

Завдяки цим механізмам у SwiftUI реалізується односпрямований потік даних: інтерфейс є відображенням (функцією) поточного стану, а будь-яка зміна стану призводить до автоматичного й узгодженого оновлення відповідних частин інтерфейсу.

Точкою входу SwiftUI-застосунку є структура, що відповідає протоколу App, у якій визначається сцена (Scene), найчастіше WindowGroup, що містить кореневий View. Навігація між екранами реалізується за допомогою компонентів NavigationStack та NavigationLink, а також модальних подань (sheets). Декларативний характер і вбудована підтримка реактивного стану роблять SwiftUI природним для застосування з архітектурним шаблоном MVVM (детальніше розглянуто у підрозділі 3.7): роль View виконують декларативні подання SwiftUI, а ViewModel реалізується як ObservableObject, що містить логіку підготовки даних для відображення.

Розробка інтерфейсу засобами SwiftUI тісно інтегрована із середовищем Xcode. Спеціальна панель попереднього перегляду Canvas дозволяє в режимі реального часу (Live Preview) бачити результат виконання коду без запуску застосунку на пристрої чи симуляторі. Зміни, внесені у програмний код, миттєво відображаються у Canvas, і навпаки – редагування елементів у візуальному режимі автоматично синхронізується з кодом. Така двостороння синхронізація та механізм «гарячого» перезавантаження значно прискорюють процес створення й налагодження інтерфейсу.

Узагальнене порівняння фреймворків SwiftUI та UIKit за основними характеристиками наведено в таблиці 4.

Таблиця 4 – Порівняльна характеристика фреймворків SwiftUI та UIKit

Характеристика	SwiftUI	UIKit
Парадигма побудови UI	Декларативна	Імперативна
Рік появи	2019	2008
Обсяг коду	Менший, компактний	Більший, шаблонний
Оновлення інтерфейсу	Автоматичне (реактивне)	Переважно ручне
Мінімальна версія iOS	iOS 13 і новіші	Усі версії
Зрілість і повнота API	Активно розвивається	Висока, перевірена часом
Попередній перегляд	Live Preview у Canvas	Обмежений

Попри активний розвиток, SwiftUI має певні обмеження: частина можливостей доступна лише у новіших версіях iOS, а деякі складні сценарії досі простіше реалізувати засобами UIKit. Тому на практиці поширеним є поєднання обох фреймворків в одному проєкті. Для інтеграції UIKit-компонентів у SwiftUI використовують протоколи UIViewRepresentable та UIViewControllerRepresentable, а для вбудовування SwiftUI-подань у UIKit-застосунок – клас UIHostingController. Це дозволяє поступово переходити на SwiftUI, зберігаючи наявний код.

Під час підготовки до цієї теми необхідно розглянути питання:

Створення візуальної частини додатків на базі декларативного фреймворку SwiftUI, зв'язування даних з елементами графічного інтерфейсу (Data Binding), інтерактивне відображення та синхронізація програмного коду з графічним інтерфейсом в xCode.

3.6 Побудова користувацького інтерфейсу в Android засобами Jetpack Compose

Мета: вивчення інструментів та засобів побудови користувацького інтерфейсу мобільних додатків для операційної системи Android засобами декларативного фреймворку Jetpack Compose.

Jetpack Compose – це сучасний інструментарій (toolkit) компанії Google для побудови користувацьких інтерфейсів нативних додатків Android, офіційно представлений у 2019 році, а у стабільній версії – у 2021 році. Він повністю базується на мові програмування Kotlin і призначений для декларативного опису інтерфейсу безпосередньо у програмному коді, без використання окремих XML-файлів розмітки. Jetpack Compose поступово стає рекомендованим підходом до розробки UI під Android, проте співіснує з традиційною системою візуальних компонентів (View) та XML-розмітки, яка залишається широко використовуваною, зокрема у наявних (legacy) проєктах.

Як і SwiftUI у середовищі Apple, Jetpack Compose реалізує декларативну парадигму побудови інтерфейсу. На відміну від традиційного імперативного підходу, де елементи описуються в XML-файлах, а потім отримуються та змінюються у коді (наприклад, через `findViewById`) вручну, у декларативному підході розробник описує, яким інтерфейс має бути для певного стану даних, а фреймворк самостійно оновлює відображення при зміні цього стану. Це зменшує обсяг шаблонного коду та усуває потребу в ручній синхронізації стану даних і елементів інтерфейсу.

В основі Jetpack Compose лежить кілька фундаментальних понять:

- композиційні функції (composable-функції) – основний будівельний блок інтерфейсу. Це звичайні функції Kotlin, позначені анотацією `@Composable`, які описують частину інтерфейсу; складніші екрани будуються через виклик одних composable-функцій усередині інших;
- композиція – інтерфейс будується шляхом поєднання та вкладення простих composable-функцій, а не через успадкування класів, що робить код модульним і придатним для повторного використання;
- єдине джерело істини та односпрямований потік даних – стан зберігається в одному визначеному місці, передається вниз по ієрархії компонентів, а події (наприклад, натискання) передаються вгору;
- рекомпозиція (recomposition) – механізм, за яким Compose автоматично й вибірково повторно викликає лише ті composable-функції, дані яких змінилися, оновлюючи відповідні частини інтерфейсу.

Побудова інтерфейсу засобами Jetpack Compose здійснюється шляхом комбінування готових composable-компонентів та контейнерів компонування. До базових елементів належать: Text (текстовий напис), Image (зображення), Button (кнопка), TextField (поле введення тексту), Icon (піктограма), Checkbox (прапорець) та інші. Для розташування елементів використовуються контейнери компонування:

- **Column** – вертикальне розташування елементів (один під одним);
- **Row** – горизонтальне розташування (один поряд з одним);
- **Box** – накладання елементів один на одного;
- **LazyColumn** та **LazyRow** – прокручувані списки з «лінивим» завантаженням лише видимих елементів;
- **LazyVerticalGrid** – сітка для ефективного відображення великих наборів даних.

Зовнішній вигляд, розміри та поведінку елементів налаштовують за допомогою параметра **Modifier** – об'єкта, методи якого утворюють послідовний ланцюжок (наприклад, задання відступів, розмірів, кольору тла, обробки натискань). Розробник також може створювати власні **composable**-функції, виділяючи повторювані фрагменти інтерфейсу, що підвищує читабельність і повторне використання коду.

Приклад простого екрана на Jetpack Compose, що містить заголовок, лічильник і кнопку для його збільшення, наведено нижче.

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }

    Column(
        verticalArrangement = Arrangement.spacedBy(20.dp)
    ) {
        Text(
            text = "Лічильник: $count",
            style = MaterialTheme.typography.titleLarge
        )

        Button(onClick = { count++ }) {
            Text("Збільшити")
        }
    }
}
```

Важливою складовою Jetpack Compose є механізм управління станом. Оскільки інтерфейс автоматично реагує на зміни стану через рекомпозицію, фреймворк надає спеціальні засоби для зберігання й відстеження стану:

- **mutableStateOf** – створює об'єкт стану, за змінами якого Compose спостерігає; зміна його значення автоматично спричиняє рекомпозицію;
- **remember** – зберігає значення (зокрема об'єкт стану) між рекомпозиціями, щоб воно не створювалося заново при кожному перемальовуванні;
- **rememberSaveable** – додатково зберігає стан при зміні конфігурації (наприклад, повороті екрана);
- **підняття стану (state hoisting)** – підхід, за якого стан виноситься з **composable**-функції до її «батька», що робить компоненти повторно використовуваними та зручними для тестування.

Для складніших застосунків стан і логіку підготовки даних зазвичай виносять у клас **ViewModel**, а дані надають у вигляді спостережуваних

потоків (StateFlow або LiveData), на які реагує інтерфейс. Завдяки цьому реалізується односпрямований потік даних: інтерфейс є відображенням поточного стану, а будь-яка його зміна призводить до узгодженого оновлення відповідних компонентів.

Точкою входу інтерфейсу Compose-застосунку є виклик функції `setContent` у методі `onCreate` активності (Activity), усередині якого визначається кореневий `composable`-вміст. Навігація між екранами реалізується за допомогою бібліотеки Navigation Compose (компоненти `NavController` та `NavHost`). Декларативний характер і вбудована підтримка реактивного стану роблять Jetpack Compose природним для застосування з архітектурним шаблоном MVVM (детальніше розглянуто у підрозділі 3.7): роль View виконують `composable`-функції, а ViewModel містить стан і логіку підготовки даних.

Розробка інтерфейсу засобами Jetpack Compose тісно інтегрована із середовищем Android Studio. Анотація `@Preview` дозволяє відображати окремі `composable`-функції на панелі попереднього перегляду без запуску застосунку на пристрої чи емуляторі. Режими Interactive Preview та Live Edit дають змогу взаємодіяти з попереднім переглядом і бачити внесені у код зміни в режимі реального часу, що значно прискорює створення й налагодження інтерфейсу.

Узагальнене порівняння Jetpack Compose та традиційної системи XML-розмітки (View) за основними характеристиками наведено в таблиці 5.

Таблиця 5 – Порівняльна характеристика Jetpack Compose та системи XML Layouts (View)

Характеристика	Jetpack Compose	XML Layouts (View)
Парадигма побудови UI	Декларативна	Імперативна
Стабільна версія	2021 рік	Доступна з ранніх версій Android
Опис інтерфейсу	Kotlin-код	XML-файли + Kotlin/Java
Обсяг коду	Менший, без <code>findViewById</code>	Більший, шаблонний
Оновлення інтерфейсу	Автоматичне (recomposition)	Переважно ручне
Доступ до елементів	Безпосередньо у коді	<code>findViewById</code> або View Binding
Попередній перегляд	<code>@Preview</code> , Live Edit	Layout Editor

Попри активний розвиток, Jetpack Compose можна поєднувати з традиційною системою View в одному проєкті, що дозволяє поступово переходити на новий підхід. Для вбудовування `composable`-вмісту в наявний інтерфейс на основі View використовують елемент `ComposeView`, а для

розміщення традиційних View-компонентів усередині Compose – функцію `AndroidView`. Це забезпечує поетапну міграцію наявного коду без потреби повного переписування застосунку.

Під час підготовки до цієї теми необхідно розглянути питання:

Створення візуальної частини додатків на базі фреймворку Jetpack Compose, інтерактивне відображення та синхронізація програмного коду з графічним інтерфейсом в Android Studio.

3.7 Сучасні архітектурні принципи та підходи до побудови додатків для мобільних пристроїв

Мета: огляд сучасних архітектурних принципів побудови програмних застосунків в контексті мобільної розробки.

У сучасній розробці додатків архітектурні принципи та шаблони (іноді також використовують терміни «архітектурний паттерн» або «архітектурний підхід», англ. software architecture pattern/framework) – це узагальнені типові рішення для організації програмного коду в процесі розробки програмного забезпечення, які допомагають структурувати логіку застосунку, полегшують його підтримку, тестування та масштабування. Зокрема, в сфері розробки додатків для мобільних пристроїв (мобільній розробці) їхнє використання особливо важливе з кількох причин:

- Розділення обов’язків (single responsibility) – використання архітектурних принципів дозволяють розділити відповідальність між різними компонентами додатка (наприклад, користувацький інтерфейс (UI), бізнес-логіка, зберігання даних, робота з мережею і т.ін.). Це знижує так звану «зв’язаність» програмного коду (тобто дає змогу логічно виділити та ізолювати окремі частини, що відповідають за певну функціональність) та полегшує його підтримку в майбутньому. Наприклад:

- UI (інтерфейс) – відповідає лише за відображення.
- Бізнес-логіка – обробляє правила та дії користувача.
- Дані – отримання, зберігання, оновлення (API, база даних тощо).

- Масштабованість (scalability) – коли архітектурні підходи допомагають зберегти структуру і надають можливість швидкого і гнучкого розширення функціональності додатка. Без неї програмний код стає складним, заплутаним, з великою кількістю повторень.

- Тестованість (testability) – коли бізнес-логіка відділена від користувацького інтерфейсу, її можна легко протестувати (наприклад шляхом написання юніт-тестів). Це важливо для якісного, стабільного продукту.

- Командна робота – добре структурований додаток дає змогу розділяти завдання між членами команди розробників, наприклад хтось займається інтерфейсом, інші – бізнес-логікою або роботою з даними.

- Підтримка та розширення (supportability) – коли застосована чітка і зрозуміла архітектура, то вносити зміни (наприклад, додати нову функцію чи змінити API) значно простіше й безпечніше, оскільки усе реалізовано згідно усім зрозумілих і погоджених правил.

- Повторне використання коду (reusability) – коли можна виділити певні частини продукту або модулі (наприклад, автентифікацію, роботу з мережею), які потім можна легко інтегрувати в інші проекти.

Нижче розглянуто лише деякі основні архітектурні підходи, що використовуються при створенні застосунків для Android та iOS, до найбільш популярних з яких можна віднести:

MVC (Model–View–Controller) – це один із найстаріших і найвідоміших патернів, який використовується для організації структури застосунку, особливо у графічних інтерфейсах користувача. Його головна ідея – розділити застосунок на три незалежні компоненти, кожен з яких відповідає за свій рівень, а саме:

- Model (Модель):
 - відповідає за дані, бізнес-логіку та правила;
 - може взаємодіяти з базами даних, API, локальним сховищем;
 - не знає нічого про інтерфейс користувача.
- View (Представлення):
 - відповідає за відображення даних користувачу;
 - це UI-елементи (екрани, кнопки, поля введення);
 - отримує дані від Controller або безпосередньо з Model.
- Controller (Контролер):
 - посередник між Model і View;
 - обробляє дії користувача (натискання кнопок, введення тексту);
 - отримує дані від Model і передає їх до View.

MVC історично використовує і просуває Apple, особливо в застосунках, основаних на використанні UIKit (на відміну від SwiftUI, де домінує MVVM підхід). В розробці під ОС Android зустрічається рідше, поступаючись вдосконаленим підходам MVP та MVVM.

MVP (Model–View–Presenter) – це вдосконалена версія MVC, яка забезпечує краще розділення обов'язків і дозволяє легше тестувати логіку застосунку. Основними компонентами MVP є:

- Model (Модель):
 - містить дані та бізнес-логіку;
 - взаємодіє з джерелами даних: бази даних, API, файлове сховище;
 - не має жодного уявлення про UI.
- View (Представлення):
 - інтерфейс користувача, який відображає інформацію;
 - пасивний компонент – не містить логіки;
 - передає дії користувача до Presenter;
 - зазвичай представлений абстрактним інтерфейсом.
- Presenter (Презентер):

- посередник між View і Model;
- обробляє дії користувача (що надходять із View);
- викликає логіку Model;
- оновлює View із результатами.

MVP широко використовується в мобільній розробці, особливо був поширений в додатках для Android до появи MVVM та Jetpack Architecture Components. Водночас, в додатках для iOS зустрічається рідко, де більш поширеним можна назвати MVC.

MVVM (Model–View–ViewModel) – один із найпопулярніших архітектурних шаблонів у сучасній мобільній розробці, особливо в Android (з використанням Jetpack) та iOS (з SwiftUI). Його головна мета – відокремити логіку представлення від інтерфейсу користувача, щоб зробити код масштабованим, тестованим і зручним у підтримці. Основними компонентами MVVM є:

- Model (Модель):
 - відповідає за бізнес-логіку та джерела даних (API, база даних тощо);
 - повністю незалежна від UI.
- View (Представлення):
 - графічний інтерфейс користувача (екрани, кнопки, списки);
 - пасивно реагує на зміни у ViewModel;
 - не містить логіки – тільки відображення.
- ViewModel (Модель представлення):
 - посередник між View і Model;
 - містить усю логіку роботи з UI (наприклад, перевірку введення, запити до моделі);
 - сповіщає View про зміни через спостерігачів (Observers).

Таким чином, MVVM є сучасним гнучким шаблоном в мобільній розробці, який добре підходить для додатків середньої та високої складності. Він дозволяє створювати структуровану архітектуру, добре організований код, особливо в поєднанні з такими технологіями як Jetpack Compose (Android) або SwiftUI (iOS).

Доцільно згадати й інші поширені сучасні підходи, зокрема архітектурний шаблон MVI (Model-View-Intent), що базується на концепції «подій», шаблон Clean Architecture – підхід до побудови програмного забезпечення, який пропонує чітке розділення відповідальностей між абстрактними шаблонами, та деякі інші.

У таблиці 3 наведено деякі поширені архітектурні шаблони, що застосовуються в розробці додатків для мобільних операційних систем iOS та Android, також порівняння деяких їхніх аспектів разом з сильними та слабкими сторонами кожного.

Таблиця 3 – Порівняльна характеристика деяких архітектурних шаблонів для розробки додатків для мобільних пристроїв

Архітектурний шаблон	Сфера використання	Переваги	Недоліки
MVC	iOS UIKit, Android legacy	Відносна простота підтримки та тестування, можливість перевикористання View або Model	Погана ізоляція (View та Controller часто тісно зв'язані або взагалі об'єднані в один об'єкт ViewController), перевантажені контролери («God object»)
MVP	Android Java/Kotlin	Висока тестованість, чітке розділення відповідальностей	Важко масштабувати, ручне оновлення UI (складна підтримка стану у View)
MVVM	Android Jetpack Compose, iOS SwiftUI	Краще розділення відповідальності (ніж у MVC або MVP), реактивність (UI автоматично оновлюється при зміні даних), легке тестування коду, вбудована підтримка у Jetpack та SwiftUI	Ускладнення структури для простих додатків, багато допоміжних класів-«обгорток»
MVI	Android Jetpack Compose, Flutter	Єдиний state, стабільність	Складніша реалізація
Clean Architecture	Великі проекти	Гнучкість, ізоляція шарів	Складність, потреба у досвіді і високій кваліфікації команди

Варто зазначити, що не існує єдиного каталогу чи переліку архітектурних шаблонів, що використовуються в розробці додатків для мобільних пристроїв. До того ж, особливості таких шаблонів можуть залежати від предметної (доменної) області, для якої розробляється програмне забезпечення, розміру проекту та досвіду команди. Більш того, навіть в рамках одного загальноприйнятого шаблону певні його аспекти чи деталі реалізації можуть значно відрізнятися для різних проектів, компаній, платформ чи мов програмування. Водночас, узагальнена категоризація архітектурних шаблонів і підходів дозволяє структурувати програмний код, описувати і використовувати світовий досвід щодо розробки програмного забезпечення, дає змогу членам команди бути в одному професійному контексті з тим самим набором термінів і т.ін.

Під час підготовки до цієї теми необхідно самостійно розглянути питання:

- Реалізація архітектурного шаблону MVC в iOS.
- Реалізація архітектурного шаблону MVP в Android.

4 ПЕРЕЛІК ПИТАНЬ ДЛЯ САМОКОНТРОЛЮ

Модуль 1. Мобільні пристрої. Додатки для мобільних ОС Android та iOS

Архітектура мобільного пристрою. Архітектура iOS та Android. Життєві цикли iOS та Android додатків

1. Які основні типи мобільних пристроїв існують (смартфони, планшети, розумні годинники тощо) та чим відрізняються їхні апаратні характеристики?
2. Назвіть основні апаратні компоненти мобільного пристрою. Яке призначення мають CPU та GPU? Чим відрізняється RAM від Flash Storage?
3. Які модулі зв'язку та сенсори типово присутні у сучасному смартфоні?
4. Перерахуйте основні програмні компоненти мобільної ОС від ядра до прикладного рівня.
5. Що таке HAL (Hardware Abstraction Layer) і яка його роль?
6. Опишіть узагальнену архітектуру ОС iOS (Core OS, Core Services, Media, Cocoa Touch).
7. Опишіть узагальнену архітектуру ОС Android. Що таке Android Runtime (ART)?
8. Опишіть життєвий цикл Activity в Android-додатку.
9. Опишіть життєвий цикл iOS-додатку.
10. Поясніть концепцію «пісочниці» (sandboxing) у мобільних ОС.
11. Як реалізовано модель безпеки та керування правами доступу (permissions) у мобільних ОС Android та iOS?
12. Назвіть спільні риси та основні відмінності платформ Android і iOS.

Сучасні підходи до створення додатків для мобільних пристроїв

13. Як класифікуються мобільні додатки (нативні, гібридні, веб-додатки) та які переваги й недоліки кожного типу?
14. Охарактеризуйте нативний, кросплатформний та гібридний підходи до розробки. Назвіть основні мови програмування для розробки під iOS та Android.
15. Які кросплатформні фреймворки існують та які мови вони використовують? Порівняйте нативну і кросплатформну розробку за продуктивністю, доступом до API та вартістю.
16. Порівняйте магазини додатків Google Play та App Store, а також підходи до монетизації (Google Play Billing vs App Store In-App Purchases).

Інструменти інтегрованого середовища програмування Xcode

17. Назвіть основні функціональні зони Xcode та їхнє призначення. Що таке Interface Builder і у якому форматі зберігаються створені у ньому інтерфейси?

18. Які платформи та мови програмування підтримує Xcode?

19. У чому різниця між тестуванням iOS-додатку на симуляторі та на реальному пристрої?

Інструменти інтегрованого середовища програмування Android Studio

20. Назвіть основні функціональні зони Android Studio та їхнє призначення. Що таке Layout Editor і чому за замовчуванням використовується ConstraintLayout? Опишіть призначення основних Tool Windows (Logcat, Profiler, Debugger, Terminal).

21. Що таке Android Virtual Device (AVD) і для чого він використовується?

Модуль 2. Розробка програмного забезпечення для мобільних пристроїв

Побудова користувацького інтерфейсу в iOS засобами SwiftUI

1. У чому різниця між імперативним і декларативним підходами до побудови UI?

2. Що таке SwiftUI та які його ключові особливості?

3. Що таке Data Binding у SwiftUI? Назвіть основні property wrappers (@State, @Binding, @ObservedObject).

4. Як забезпечується синхронізація коду SwiftUI з графічним інтерфейсом у Xcode (Live Preview / Canvas)?

5. Як створюються власні (custom) View у SwiftUI?

6. Порівняйте SwiftUI та UIKit за стилем коду, реактивністю і складністю опанування.

Побудова користувацького інтерфейсу в Android засобами Jetpack Compose

7. Що таке Jetpack Compose та яка його роль у сучасній Android-роботі?

8. Що таке Composable-функції? Як використовується анотація @Composable?

9. Поясніть концепцію стану (state) та recomposition у Jetpack Compose.

10. Як забезпечується синхронізація коду Compose з графічним інтерфейсом у Android Studio (Preview / Live Edit)?

11. Назвіть основні стандартні компоненти Jetpack Compose (Text, Button, Column, Row, LazyColumn).

12. Порівняйте підходи XML Layouts і Jetpack Compose: переваги, недоліки, сценарії застосування.

Сучасні архітектурні принципи та підходи до побудови додатків для мобільних пристроїв

13. Що таке архітектурний шаблон і яке його призначення?
14. Назвіть та коротко поясніть основні переваги застосування архітектурних принципів (single responsibility, scalability, testability, reusability).
15. Опишіть структуру шаблону MVC: компоненти та зв'язки між ними.
16. Опишіть структуру шаблону MVP: компоненти та зв'язки між ними.
17. Опишіть структуру шаблону MVVM: компоненти та зв'язки між ними.
18. У чому полягає принципова відмінність між MVC і MVP, а також між MVP і MVVM?
19. На яких платформах та з якими технологіями типово застосовуються MVC, MVP та MVVM?
20. Що таке шаблон MVI та на якій концепції він базується?
21. Що таке Clean Architecture та які її основні переваги і недоліки?
22. Чому MVVM добре поєднується з SwiftUI та Jetpack Compose?
23. Як архітектурний шаблон впливає на тестованість додатку та організацію командної роботи?

5. ТЕСТОВІ ЗАВДАННЯ ДЛЯ САМОКОНТРОЛЮ

Оберіть одну або декілька правильних відповідей. Правильні відповіді наведено у ключі в кінці розділу.

1. Що є основним обчислювальним блоком мобільного пристрою, який виконує інструкції операційної системи та додатків?

- A) Центральний процесор (CPU)
- B) Графічний процесор (GPU)
- C) Оперативна пам'ять (RAM)
- D) Постійна пам'ять (Flash Storage)

2. Які з наведених компонентів належать до апаратної частини мобільного пристрою?

- A) Графічний процесор (GPU)
- B) Акселерометр
- C) Android Runtime (ART)
- D) Літій-іонний акумулятор
- E) Application Framework

3. Який програмний шар забезпечує уніфікований доступ до апаратного забезпечення без знання деталей його реалізації?

- A) Ядро (Kernel)
- B) Апаратна абстракція (HAL)
- C) Application Framework
- D) Користувацькі додатки

4. На чому базується ядро операційної системи Android?

- A) На XNU
- B) На Linux
- C) На Windows NT
- D) На FreeBSD

5. Який метод викликається першим у життєвому циклі Activity в Android?

- A) onStart()
- B) onResume()
- C) onCreate()
- D) onInit()

6. Що таке «пісочниця» (sandboxing) у мобільних операційних системах?

- A) Інструмент тестування інтерфейсу
- B) Механізм ізоляції додатків один від одного та від системи
- C) Режим енергозбереження
- D) Сховище тимчасових файлів

7. Які мови програмування є основними для нативної розробки під iOS?

- A) Swift
- B) Objective-C
- C) Kotlin
- D) Java

8. Які мови програмування є основними для нативної розробки під Android?

- A) Kotlin
- B) Java
- C) Swift
- D) Objective-C

9. Яка мова програмування використовується у фреймворку Flutter?

- A) Java
- B) Kotlin
- C) Dart
- D) Swift

10. Які з наведених фреймворків призначені для кросплатформної розробки?

- A) Flutter
- B) React Native
- C) UIKit
- D) Jetpack Compose

11. Що характеризує гібридний підхід до розробки мобільних додатків?

- A) Окремий код для кожної платформи рідними мовами
- B) Запуск вебдодатку всередині веб-контейнера (WebView)
- C) Повна інтеграція з апаратним забезпеченням
- D) Найвища продуктивність серед усіх підходів

12. Які платформи підтримує середовище розробки Xcode?

- A) iOS
- B) macOS
- C) Android
- D) watchOS
- E) Windows

13. Який інструмент Xcode призначений для візуального створення графічного інтерфейсу?

- A) Project Navigator
- B) Interface Builder
- C) Debug area
- D) Issue Navigator

14. У якому форматі зберігаються інтерфейси, створені за допомогою Interface Builder?

- A) .swift
- B) .json
- C) .xib (на основі XML)
- D) .txt

15. На основі якого середовища побудовано Android Studio?

- A) Eclipse
- B) IntelliJ IDEA
- C) Visual Studio
- D) NetBeans

16. Який файл описує компоненти Android-додатку та дозволи (permissions)?

- A) build.gradle
- B) MainActivity.kt
- C) AndroidManifest.xml
- D) styles.xml

17. Який контейнер використовується за замовчуванням у Layout Editor середовища Android Studio?

- A) LinearLayout
- B) FrameLayout
- C) ConstraintLayout
- D) RelativeLayout

18. Який інструмент Android Studio відображає журнал подій додатку під час його виконання?

- A) Logcat
- B) Layout Editor
- C) Gradle
- D) Status bar

19. До якої парадигми побудови інтерфейсу належить фреймворк SwiftUI?

- A) Імперативної
- B) Декларативної
- C) Процедурної
- D) Логічної

20. Який property wrapper у SwiftUI використовується для зберігання локального змінюваного стану всередині View?

- A) @State
- B) @Binding
- C) @EnvironmentObject
- D) @Published

21. Які контейнери компонування використовуються у SwiftUI?

- A) VStack
- B) HStack
- C) ZStack
- D) Column

22. Який засіб Xcode дозволяє бачити результат коду SwiftUI у реальному часі без запуску на пристрої?

- A) Logcat
- B) Canvas (Live Preview)
- C) Interface Builder
- D) Layout Editor

23. Якою анотацією позначаються композиційні функції у Jetpack Compose?

- A) @Override
- B) @Composable
- C) @Preview
- D) @State

24. Як називається механізм автоматичного оновлення інтерфейсу при зміні стану у Jetpack Compose?

- A) Компіляція
- B) Реконпозиція (recomposition)
- C) Інфляція (inflation)
- D) Сериалізація

25. Які контейнери компоновання використовуються у Jetpack Compose?

- A) Column
- B) Row
- C) Box
- D) VStack

26. Які елементи забезпечують сумісність Jetpack Compose із традиційною системою View?

- A) ComposeView
- B) AndroidView
- C) findViewById
- D) UIHostingController

27. Які компоненти входять до архітектурного шаблону MVC?

- A) Model, View, Controller
- B) Model, View, Presenter
- C) Model, View, ViewModel
- D) Model, View, Intent

28. Яка роль компонента View у шаблоні MVVM?

- A) Містить бізнес-логіку та джерела даних
- B) Пасивно реагує на зміни у ViewModel і відображає дані
- C) Безпосередньо звертається до бази даних
- D) Обробляє лише мережеві запити

29. Які архітектурні шаблони найчастіше застосовуються у сучасній мобільній розробці?

- A) MVC
- B) MVP
- C) MVVM
- D) HTML

30. Що є основною перевагою застосування архітектурних шаблонів у розробці мобільних додатків?

- A) Збільшення обсягу коду
- B) Полегшення підтримки, тестування та масштабування
- C) Відмова від використання IDE
- D) Гарантована сумісність з усіма версіями ОС

Ключ до тестових завдань

1 – A; 2 – A, B, D; 3 – B; 4 – B; 5 – C; 6 – B;
7 – A, B; 8 – A, B; 9 – C; 10 – A, B; 11 – B; 12 – A, B, D;
13 – B; 14 – C; 15 – B; 16 – C; 17 – C; 18 – A;
19 – B; 20 – A; 21 – A, B, C; 22 – B; 23 – B; 24 – B;
25 – A, B, C; 26 – A, B; 27 – A; 28 – B; 29 – A, B, C; 30 – B.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Положення про організацію самостійної роботи здобувачів вищої освіти у Вінницькому національному технічному університеті. Вінниця, 2024. 16 с. URL: <https://vntu.edu.ua/uploads/n/np/4.pdf> (дата звернення: 12.04.2026).
2. Мартін Р. Чистий код. Створення і аналіз / пер. з англ. Харків : Фабула, 2019. 464 с.
3. Мартін Р. Чиста архітектура / пер. з англ. Харків : Фабула, 2019. 368 с.
4. Фаулер М. Рефакторинг. Покращення наявного коду / пер. з англ. Київ : Наш Формат, 2021. 448 с.
5. Murphy M. The Busy Coder's Guide to Android Development. CommonsWare, 2023. 2500 p.
6. Horton J. Android Programming with Kotlin for Beginners: Build Android apps starting from zero programming experience. 3rd ed. Packt Publishing, 2023. 770 p.
7. Griffiths Dawn, Griffiths David. Head First. Android Development: A Learner's Guide to Building Android Apps with Kotlin. 3rd ed. O'Reilly Media, 2021. 930 p.
8. Phillips B., Stewart Ch., Marsicano K. Android Programming: The Big Nerd Ranch Guide. 5th ed. Big Nerd Ranch Guides, 2022. 624 p.
9. Sahar A. iOS 18 Programming for Beginners: Learn iOS development with Swift 6, Xcode 16, and iOS 18 - your path to App Store success. 9th ed. Packt Publishing, 2024. 584 p.
10. Kunneth T. Android UI Development with Jetpack Compose: Bring declarative and native UI to life quickly and easily on Android using Jetpack Compose and Kotlin 2nd ed. Edition. Packt Publishing, 2023. 278 p.
11. Moskala M. Android Development with Kotlin. Packt Publishing, 2022. 508 p.
12. Späth P. Learn Kotlin for Android Development : The Next Generation. 2019. 801 p.
13. Радченко К. О. Розроблення мобільних застосунків. Київ : КПІ ім. Ігоря Сікорського, 2023. 546 с.
14. Сьерра К, Бейтс Б. Head First Java. Легкий для сприйняття довідник / пер. з англ. Харків : Фабула, 2022. 720 с.
15. Gerardus Blokdyk Network computer operating system. 5STARCooks, 2021. 317 p.
16. Tanenbaum A. S. Computer architecture – Strukturen, Konzepte, Grundlagen. 6th ed. Pearson Education, 2016. 829 p.
17. Tanenbaum A. S. Structured computer organization. Todd Austin. 6th ed. 2018. 801 p.

18. Азаров О. Д., Захарченко С. М., Кадук О. В., Орлова М. М. Комп'ютерні мережі : підручник. Вінниця : ВНТУ, 2020. 378 с.

19. Азаров О. Д., Муращенко О. Г., Черняк О. І. Генерування високо-лінійних пілкоподібних сигналів на базі ЦАП з низькогітчевим кодуванням. Вінниця : ВНТУ, 2021. 132 с.

20. Романько Р. О., Городецька О. С., Богомолів С. В. Оптимізація мобільних застосунків за допомогою фреймворку React Native. *Матеріали LIII науково-технічної конференції підрозділів ВНТУ*, м. Вінниця, 20-22 берез. 2024 р. Вінниця, 2024. URL: <https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2024/paper/view/19953/16520> (дата звернення: 28.04.2026).

Інформаційні ресурси

21. Apple Developer Portal. URL: <https://developer.apple.com/documentation/> (дата звернення: 10.05.2026).

22. Android Developer Portal. URL: <https://developer.android.com/> (дата звернення: 10.05.2026).

23. Мова програмування Swift. URL: <https://book.swift.org.ua> (дата звернення 10.05.2026).

24. Google Pathways. URL: <https://developers.google.com/learn/pathways> (дата звернення 10.05.2026).

Електронне навчальне видання

**Олександр Геннадійович Муращенко
Оксана Степанівна Городецька**

**Методичні вказівки до виконання самостійної роботи з
дисципліни «Проектування додатків для мобільних
пристроїв» зі спеціальності «Комп'ютерна інженерія»**

Рукопис оформила: *О. Городецька*

Редактор: *Н. Кравчук*

Оригінал-макет виготовлено в РВВ ВНТУ

Підписано до видання 31.07.2026

Гарнітура Times New Roman.

Зам. № Р2026-104.

Видавець та виготовлювач

Вінницький національний технічний університет,

Редакційно-видавничий відділ.

ВНТУ, ГНК, к. 114.

Хмельницьке шосе, 95,

м. Вінниця, 21021.

press.vntu.edu.ua;

Email: rvv.vntu@gmail.com.

Свідоцтво суб'єкта видавничої справи
серія ДК No 3516 від 01.07.2009 р.