Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

Кваліфікаційна робота

освітній ступінь — бакалавр

на тему: «**Автоматичний аналіз рівня енергоспоживання мобільних застосунків**»

Виконала: студентка 4-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Грисюк Анастасія

Керівник Франків О. О.

старший викладач

«26» травня 2025 р.

Київ — 2025

Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра інформатики
Освітній ступінь бакалавр
Спеціальність 121 «Інженерія Програмного Забезпечення»
Освітня програма бакалавр

ЗАТВЕРДЖУЮ
Завідувач кафедри інформатики
Гороховський С. С.

“10” жовтня 2024 року

ЗАВДАННЯ
ДЛЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ СТУДЕНТЦІ
Грисюк Анастасії

1. Тема роботи **«Автоматичний аналіз рівня енергоспоживання мобільних застосунків»**, керівник роботи Франків Олександр Олександрович, старший викладач

2. Строк подання студентом роботи 26 травня 2025

3. План роботи

Анотація

Вступ

Розділ 1. ТЕОРЕТИЧНІ ОСНОВИ

1.1 Загальний опис

1.2 Огляд наявних статичних аналізаторів енергоспоживання мобільних застосунків

1.3 SwiftSyntax як інструмент аналізу структури коду

1.4 Висновки до розділу

Розділ 2. ДЕТАЛЬНИЙ ОГЛЯД АНТИПАТЕРНІВ

2.1 Робота з локацією

2.2 Використання таймера

2.3 Робота з Bluetooth

2.4 Використання GPU

2.5 Використання CPU

ГРАФІК ПІДГОТОВКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

№ з/п	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника	Примітки
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника Узгодження календарного графіка підготовки кваліфікаційної роботи. Ознайомлення студента з критеріями оцінювання кваліфікаційної роботи (п. 8.5).	30 вересня 2024			
2.	Вивчення джерел літератури, матеріалів архівів, періодичних видань, збір та узагальнення фактів, даних	30 вересня 2024 – 05 листопада 2024			
3.	Складання плану кваліфікаційної роботи та узгодження з науковим керівником	05 листопада 2024			
4.	Написання розділів роботи	05 листопада 2024 – 02 березня 2025			
5.	Проміжний контроль виконання роботи	01 лютого 2025			
6.	Написання кваліфікаційної роботи в цілому, ознайомлення з її першим варіантом наукового керівника	10 січня 2025 – 25 березня 2025			
	Розділ 1 (постановка проблеми, теоретичні основи, огляд літературних джерел)	20 січня 2025			
	Розділ 2 (аналітично-дослідницька частина)	27 лютого 2025			
	Розділ 3 (проєктно-рекомендаційна частина)	25 березня 2025			
7.	Повне завершення написання кваліфікаційної роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	02 квітня 2025 – 07 травня 2025			
8.	Подання кваліфікаційної роботи для перевірки письмових робіт студентів НаУКМА на відповідність вимогам академічної доброчесності,	24 травня 2025			
9.	Публічний захист кваліфікаційної роботи перед екзаменаційною комісією	згідно з розкладом роботи ЕК			

Графік узгоджено 30 вересня 2024 р.

Науковий керівник Франків Олександр Олександрович

Виконавиця кваліфікаційної роботи Грисюк Анастасія

ЗМІСТ

АНОТАЦІЯ

У цій кваліфікаційній роботі розглянуто створення інструменту для статичного аналізу коду мобільних застосунків під iOS із метою оцінки потенційного рівня їх енергоспоживання. На основі огляду наукових досліджень було визначено набір антипатернів, що негативно впливають на енергоефективність. Ці антипатерни стали основою для формалізації правил, за якими проводиться аналіз. Особливу увагу приділено реалізації статичного аналізатора за допомогою бібліотеки SwiftSyntax, яка дає змогу працювати з кодом у вигляді абстрактного синтаксичного дерева.

У результаті роботи розроблено статичний аналізатор для виявлення енергомістких частин коду під час компіляції. Такий інструмент сприятиме створенню більш енергоефективних мобільних застосунків.

Ключові слова: Swift, iOS, SwiftSyntax, ArgumentParser, статичний аналіз коду, енергоспоживання, мобільні застосунки.

ВСТУП

У сучасному світі мобільні телефони стали невіддільною частиною повсякденного життя людини. Вони використовуються не лише для зв’язку, а й для роботи, відпочинку та доступу до інформації. Попри високу функціональність та зручність, мобільні пристрої залишаються обмеженими в автономній роботі від акумулятора. На додачу до цього, часто можливість підзарядки не є доступною завжди.

Згідно з опитуванням, проведеним у рамках дослідження [1], 80% користувачів вживають заходів для збільшення життєвого часу батареї, що свідчить про брак тривалості автономної роботи пристроїв. Відповідно, питання оптимізації використання ресурсів пристрою з боку програмного забезпечення є гострим і потребує подальшого вивчення.

Ерве Ґіхо (Hervé Guihot) у книзі «Pro Android Apps Performance Optimization» [2] зазначив: *«Користувачі, як правило, не помічають, чи економить ваш застосунок час роботи від батареї. Однак вони, швидше за все, помітять, якщо це не так. Оскільки користувачі часто видаляють застосунки, які споживають занадто багато енергії, ваш застосунок повинен розумно використовувати батарею, але при цьому надавати користувачам можливість налаштовувати певні функції, оскільки різні користувачі будуть просити про різні речі. Надайте користувачам більше можливостей».* Попри швидкі процесори та мережі, а також великий обсяг пам'яті, користь смартфонів обмежується часом роботи від батареї. Як наслідок, енергоефективність смартфонів посідає важливе місце серед напрямів дослідження мобільних застосунків.

Упродовж тривалого часу інженери не переймалися енергоефективністю, а зосереджувалися на таких показниках якості, як відсутність помилок, продуктивність та надійність. Завдання покращення енергоефективності залишили розробникам операційних систем. Дійсно, дослідникам вдається створювати дедалі енергоефективніші комп’ютерні архітектури. Поза тим,

нещодавні емпіричні дослідження надали докази того, що більших результатів можна досягти, заохотивши розробників програмного забезпечення відігравати не менш важливу роль у зменшенні енергоспоживання завдяки високорівневому проєктуванню та коректній імплементації необхідного функціоналу [4].

Протягом останніх декад небагато уваги приділено створенню технік та інструментів, які дали б змогу розробникам краще розуміти та використовувати ресурси батареї. Як наслідок, розробникам бракує детальних інструкцій, курсів та інструментів, до яких можна звернутися з метою покращити своє розуміння того, як писати енергоефективне програмне забезпечення [3].

Зважаючи на вищезазначену інформацію, темою цієї кваліфікаційної роботи обрано автоматичний аналіз рівня енергоспоживання мобільних застосунків.

Об’єктом дослідження стало енергоспоживання мобільних застосунків на операційній системі iOS.

Предметом дослідження є методи статичного аналізу коду мобільних застосунків на Swift для виявлення антипатернів, що призводять до підвищеного енергоспоживання.

Безпосередньою **метою кваліфікаційної роботи є** створити статичний аналізатор, що знаходитиме енергомісткі практики в розробці мобільних застосунків мовою Swift.

Задля досягнення мети поставлено наступні **завдання**:

- опрацювати наявні дослідження використання енергії мобільних застосунків;
- формалізувати виявлені антипатерни у вигляді чітких правил, за якими відбуватиметься синтаксичний аналіз коду;
- опрацювати технічну літературу для вибору інструментів для написання статичного аналізатора;

- розробити інструмент для статичного аналізу;
- інтегрувати розроблений інструмент як плагін у середовище розробки XCode.

Перший розділ присвячений створенню комплексної теоретичної бази для розробки аналізатора енергоспоживання для мобільних додатків. У ньому викладено теоретичні основи, необхідні для досягнення мети. У другому розділі описано дослідження антипатернів у розробці мобільних застосунків, які є причиною надмірного споживання енергії. Для кожного виявленого антипатерну наведено ретельне пояснення, підкріплене відповідними дослідженнями, а також практичні приклади, що демонструють як неефективний, так і оптимізований код. Зіставлення початкового та виправленого коду створює підґрунтя для формалізації шкідливих практик у вигляді правил, що використовуватимуться у статичному аналізаторі. У третьому розділі мова йде про процес створення аналізатора за допомогою бібліотеки SwiftSyntax. Його можливо запускати з командного рядка, а також інтегрувати в XCode за допомогою Swift Plugin.

Результатом роботи став розроблений інструмент для статичного аналізу енергоспоживання мобільних застосунків. Ця робота сприяє раціональному використанню ресурсів пристрою та продовженню терміну служби батареї, даючи змогу заощаджувати заряд батареї без надмірного обмеження функціональності застосунку.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ

1.1 Загальний опис

З огляду на обмеженість ресурсів смартфонів, постає потреба в нових підходах до програмування, які враховують важливість ощадливого використання енергоспоживання. Застосунок, що швидко витрачає заряд енергії, призводить до незадовільного користувацького досвіду, негативних відгуків та навіть його видалення.

З метою підвищення енергоефективності програмного забезпечення ще на етапі його розробки з'явився підхід, відомий як Green Coding. Це практика, суть якої полягає в тому, аби створювати програмне забезпечення з мінімальним впливом на зовнішнє середовище. Його особливість полягає у врахуванні енергоспоживання ще на етапі розробки програмного забезпечення, позбуваючися патернів, що сприяють невиправданому розрядженню батареї.

Згідно з [5], Green Coding складається з кількох ключових принципів. Ця робота покладається на один із них, а саме Adherence to best coding practices, адже якість програмного забезпечення напряму впливає на продуктивність пристроїв. Відповідно, від застосунків залежить те, скільки енергії споживатиме пристрій. Таким чином Green Coding поступово перетворюється на важливий компонент сучасного процесу розробки застосунків. Ця практика дає змогу зменшити навантаження на ресурси пристроїв та продовжити час автономної роботи пристроїв, що сприяє меншому використанню електроенергії. У довгостроковій перспективі це відкриває можливість поєднувати функціональність із турботою про навколишнє середовище.

Одним із інструментів, що сприяє Green Coding, є аналізатори, які дають змогу знаходити шматки коду, що можуть негативно впливати на енергоефективність застосунку. Існує два підходи до аналізу програмного коду: динамічний та статичний. Динамічний відбувається під час роботи застосунку, через що його складніше інтегрувати у процес розробки. Натомість статичний

аналіз проводиться ще до запуску застосунку шляхом дослідження вихідного коду. З цієї причини статичний аналіз краще підходить для ранніх етапів розробки.

1.2 Огляд наявних статичних аналізаторів енергоспоживання мобільних застосунків

Під час ознайомлення з наявними науковими не було знайдено жодного готового аналізатора коду застосунків на платформі iOS. Утім, готових подібних інструментів для аналізу коду застосунків на платформі Android існує чимало.

1.2.1 Статичний лінтер на PMD та Adroid Lint

У роботі [23] описано створення автоматизованого лінтера для Android, що поєднує можливості Android Lint та PMD — статичного аналізатора коду, який створює звіт проблем, знайдених у коді програмного забезпечення. Механізм роботи є наступним: спершу PMD аналізує Java-код на основі дерева синтаксичного аналізу, після чого Android Lint перевіряє XML та Java-файли на наявність визначених антипатернів. Плагін для Eclipse автоматично виконує автоматичне або напівручне виправлення. Інструмент перевіряє відповідність застосунку 49 правилам оптимізації коду.

Хоч кількість цих правил є значною, усі вони зосереджені виключно на загальних принципах ефективного програмування: уникнення зайвих обчислень, мінімізація створення об’єктів, спрощення логіки роботи циклів, умов, потоків, відсутність непотрібних змінних, методів тощо. Жодне з правил не стосується роботи зі складнішими підсистемами пристрою, таких як: робота з локацією, сенсорами, графікою, Bluetooth. Це підкреслює, що навіть на рівні синтаксису можливо суттєво зменшити навантаження на пристрій.

Окрім того, деякі з цих правил є специфічними саме для Java, як, до прикладу UseStringTokenizer, UseDataSourceInsteadOfDriverManager, оскільки у Swift робота з розбиттям рядків та базами даних відбувається в інший спосіб.

Певні правила з цієї таблиці є вбудованими в компілятор Swift, зокрема рекомендація прибирати змінні, що не використовуються.

На рисунку 1.1 зображено статистику використання центрального процесора пристроюю HTC One XL для кожного правила.

Application	Rule	Before optimization	After optimization
Estate	AvoidEmptyIf	0.28/0.22/0.14	0.27/0.19/0.14
Estate	AvoidMethodCallsInLoop	0.34/0.13/0.12	0.08/0.07 /0.10
Who is millionare	UseStringLength-CompareEmptyString	0.51/0.17/0.08	0.09/0.10 /0.08
Estate	Avoid object instantiation in frequently executed code	0.38/0.12/0.08	0.32/0.11 /0.07
Book Management	A constant expression can be evaluated	0.43/0.12/0.08	0.11/0.09 /0.07
Book Management	DefineInitialCapacities	0.14/0.09/0.06	0.0 /0.04 /0.05
Who is millionare	AvoidSynchronized-methods in loop	0.22/0.07/0.06	0.01/0.08 /0.07

Рис. 1.1. Середній відсоток використання процесора за останні 1 хвилину, 5 хвилин та 10 хвилин)

Отже, в результаті вище зазначеної роботи створено інструмент, який допомагає виявляти й усувати енергомісткий код ще до виконання застосунку. Серед функціональних можливостей систем є автоматичне виявлення порушень визначених правил, що збираються та записуються в окремий звіт роботи аналізатора. Наприкінці роботи створений аналізатор інтегровано в середовище розробки Eclipse та проведено порівняльний аналіз споживання % CPU до та після оптимізації коду.

1.2.2 Static application analysis detector (SAAD)

Робота [25] присвячена виявленню енергетичних багів у застосунках на Android за допомогою статичного аналізу. Автори представили інструмент, який дає змогу знаходити два типи енергозатратних патернів: витік ресурсів та дефекти розмітки.

Витік ресурсів виникає внаслідок несвоєчасного завершення енергомістких процесів, як-от робота з GPS, камерою, сенсорами тощо. Дефекти розмітки переважно є наслідком поганого дизайну структури розмітки.

Найчастіше вони є наслідком використання глибоких вкладень View: зі збільшенням кількості об’єктів, вкладених один в один, складність макета файлу зростає.

Конкретні правила перевірки коду не описано, утім при розробці статичного аналізатора енергоспоживання мобільних застосунків варто звернути увагу на витік ресурсів, оскільки ця проблема є поширеною незалежно від мови програмування та платформи, для якої розроблено застосунок.

Фреймворк приймає на вхід APK-файл (формат архівних файлів-застосунків для Android), після чого SAAF аналізує застосунок на наявність витоку ресурсів, а Lint займається пошуком дефектів розмітки. Отримані дані про дефекти генеруються у два окремі звіти. Огляд логіки системи виявлення помилок зображено на Рисунку 1.2.

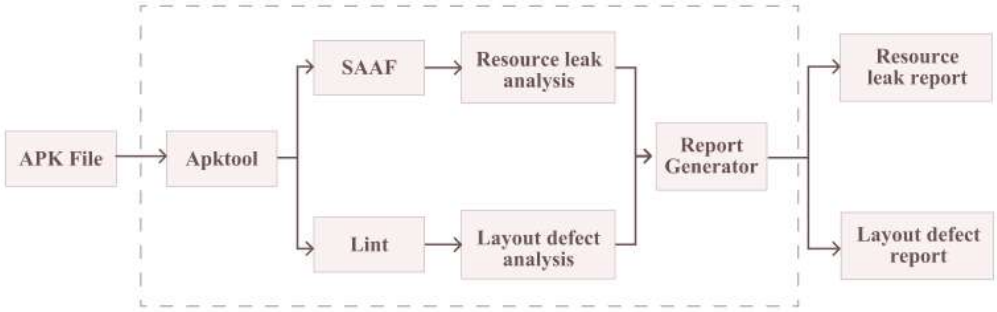


Рис. 1.2. Схема роботи аналізатора.

Створений фреймворк було протестовано на 64 застосунках, із яких 52 мали принаймні один витік ресурсів, а 45 — дефекти розмітки. Точність перевірки витоку ресурсів складає 86.67%, а дефектів розмітки — 76.27%. Отже, як витік ресурсів, так і дефекти розмітки є поширеними помилками розробників при написанні коду.

1.3 SwiftSyntax як інструмент аналізу структури коду

1.3.1. Визначення та загальний опис

У розробці програмного забезпечення надзвичайно важливими є інструменти, які дають змогу виконувати аналіз коду до виконання програми,

аби виявляти потенційні проблеми на ранніх етапах. Для мови програмування Swift таким інструментом є бібліотека SwiftSyntax, яка надає структури даних та алгоритми для аналізу Swift-коду з високорівневим, безпечним та ефективним API. SwiftSyntax — це основа, на якій побудовані такі інструменти, як синтаксичний аналізатор Swift, swift-format та Swift-макроси.

Завдяки цій бібліотеці можливо аналізувати та змінювати код, що відкриває можливості для створення інструментів розробки, підвищення якості програмного забезпечення та автоматизації багатьох рутинних завдань.

Серед основних можливостей SwiftSyntax варто виділити побудову інструментів для статичного аналізу коду, аби виявити потенційні помилки, антипатерни та недотримання стилю. Важливою є можливість автоматично рефакторити код, який порушує задані правила. За допомогою цієї бібліотеки також можна генерувати код. Зручним є можливість інтегрувати створені інструменти в XCode для виведення попереджень або ж помилок безпосередньо в середовищі розробки.

Для статичного аналіз SwiftSyntax перетворює програмний код на абстрактне синтаксичне дерево.

1.3.2. Абстрактне синтаксичне дерево

Абстрактне синтаксичне дерево (AST) є ключовою концепцією в аналізі коду. Це структура даних, у якій кожен вузол відповідає певній конструкції мови програмування: змінній, функції, класу тощо. Завдяки AST можливо зрозуміти логічну структуру програми незалежно від специфіки форматування.

Простий Swift-код let result = a + b буде представлений у вигляді дерева, як зображено на Рисунку 1.3.

Рис. 1.3. Приклад представлення коду у вигляді синтаксичного дерева

Дерево побудоване таким чином, аби відображати відносини між вкладеними синтаксичними елементами та їхніми сусідами. Воно також містить так звані trivia — несинтаксичні сутності, як-от пробіли чи коментарі, що дає змогу відновити початковий документ з повною точністю. Окрім структури, синтаксичне дерево також відображає два можливі типи помилок: синтаксис, якого бракує та неочікуваний синтаксис.

Дерево складається з елементів, що мають назву синтаксичні вузли. Кожна така вузол має покликання на батьківський вузол та вузли-нащадки. Для роботи з вузлами існує ієрархія протоколів. На вершині ієрархії знаходиться SyntaxProtocol, під яким знаходяться такі протоколи, як DeclSyntaxProtocol, ExprSyntaxProtocol, PatternSyntaxProtocol, TypeSyntaxProtocol та StmtSyntaxProtocol. Кожен із цих протоколів відповідає за окремий синтаксис: класи, виклики функцій, патерни, типи та умовні оператори.

На найнижчому рівні дерева, листках, знаходяться токени (TokenSyntax), що відображають одиницю синтаксису та асоційовану з ним leading та trailing trivia, до прикладу, ідентифікатор та пробіли навколо нього.

1.3.2. Патерн «Відвідувач»

Одним із ключових механізмів обробки абстрактного синтаксичного дерева у SwiftSyntax є використання шаблону «Відвідувач». Це поведінковий патерн проєктування, за допомогою якого можливо додавати нові операції до програми, не змінюючи класи об’єктів, над якими можуть виконуватися ці операції.

Для реалізації патерна необхідно визначити окремий відвідувач, що реалізує операцію, яка виконується над елементами структури об’єктів. Кожен елемент має метод accept(visitor), який приймає об’єкт, який знає, як його обробити. Коли програма проходить по структурі та викликає цей метод, елемент передає керування відвідувачу, який, зі свого боку, викликає відповідний метод, як-от visitFunction чи visitStatement та виконує потрібну дію над елементом.

Для проходження програмного коду з кореня до його листків у SwiftSyntax використовується SyntaxVisitor. Для дослідження кожного типу синтаксичного вузла необхідно перевизначити visit метод, який приймає вузол як параметр. Ці методи використовують in-order traversal, проте можливо також скористатися post-order traversal: для цього необхідно використовувати відповідний visitPost метод. У такий спосіб, кожен конкретний відвідувач відповідає за виявлення одного або кілького шаблонів коду. До прикладу, один відвідувач може шукати надмірне створення об’єктів, інший — назви змінних, що викликають конкретний метод, а третій — тіло функцій, які містять певну кількість аргументів.

Важливою перевагою цього шаблону, зокрема під час статичного аналізу великих проєктів, де повне дерево коду може мати тисячі елементів, є можливість фільтрації вузлів, що дає змогу уникнути надмірного відвідування непотрібних структур.

1.4 Висновки до розділу

У цьому розділі розглянуто значення енергоефективності в сучасній розробці програмного забезпечення, роль розробників у зменшенні споживання

енергії. Особливу увагу приділено наявним статичним аналізаторам енергоспоживання для Android. Попри те, що ці інструменти не мають прямих аналогів для платформи iOS, їхній аналіз став першим кроком для створення такого лінтера.

Особливу увагу приділено можливостям використання SwiftSyntax — бібліотеки для аналізу коду мовою Swift шляхом побудови абстрактного синтаксичного дерева.

У такий спосіб, сформоване розуміння як загальних принципів, так і конкретних реалізацій аналізаторів енергоспоживання створило ґрунтовну теоретичну базу для розробки власного інструменту.

Наступний розділ присвячено розгляду шкідливих практик коду, що негативно впливають на енергоспоживання мобільних пристроїв, а також формуванню правил, за якими відбуватиметься пошук антипатернів.

РОЗДІЛ 2. АНТИПАТЕРНИ НАДМІРНОГО ВИКОРИСТАННЯ РЕСУРСІВ

Згідно з офіційними даними від Apple [2], основними споживачами енергії в мобільних пристроях є центральний процесор (CPU), графічний процесор (GPU), використання локації (GPS) й таймера, а також Bluetooth.

У ході опрацювання наявних досліджень не було знайдено жодного повноцінного дослідження шкідливих патернів написання коду мовою Swift, яке б визначило бодай кілька причин швидкого розрядження батареї iPhone. Утім наукових робіт на ідентичну тему для платформи Android існує чимало.

Зважаючи на наведений вище факт, частину антипатернів взято з досліджень енергоспоживання застосунків на Android та адаптовано під iOS.

2.1 Робота з локацією

2.1.1. Визначення та загальний опис

GPS — це функція, що відповідає за відстеження місцеперебування смартфона за допомогою супутникових даних. Вона широко використовується програмами та службами ОС для надання персоналізованих рекомендацій на основі даних про локацію. Служба визначення розташування важлива, але є однією з найбільш енерговитратних служб смартфона [7].

Як помітно на Рисунку 2.1, ввімкнений GPS на пристрої розряджає батарею значно швидше [8].

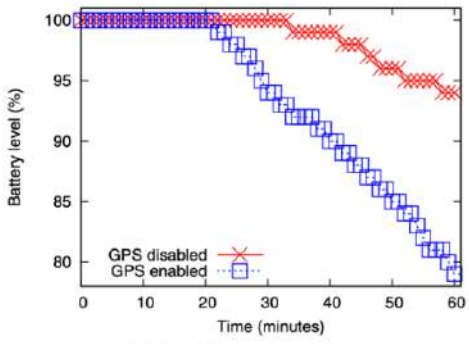


Рис. 2.1. Різниця енергоспоживання з увімкненим та вимкненим GPS.

Програми запитують розташування користувача, аби реєструвати фізичну активність осіб. Використання енергії зростає з підвищенням точності та частоти запитів на визначення локації [2].

В iOS робота з локацією відбувається за допомогою фреймворку Core Location. Для налаштування, старту та зупинки сервісів локації необхідно використовувати екземпляр класу CLLocationManager.

Існує 3 види сервісів для доставлення локації. Переважно використовується standard. Цей сервіс найкраще підходить для застосунків, що потребують точного відстеження маршруту в реальному часі. Згідно з інформацією, вказаною в документації, є найбільш точним та енергозатратним. Другий сервіс має назву Significant-change. Його доцільно використовувати у випадках, коли сповіщення про зміну локації потрібні лише при суттєвому переміщенні пристрою, до прикладу, переїзд до іншого району міста. Також вказаний як low-power. Третій сервіс, Visit, є ідеальним вибором для застосунків, яким потрібно отримувати сповіщення, коли користувач прибуває до певного місця, до прикладу, дім, робота, навчальний заклад тощо, проводить там певний час та залишає його. Він є найбільш енергоефективним [6].

2.1.2. Високоточне оновлення локації

У CLLocationManager для налаштування точності та частоти оновлення місця є два основні параметри, що дають змогу гнучко керувати процесом отримання геоданих під час розробки.

Перша властивість — desiredAccuracy. Вона повідомляє системі необхідний рівень точності визначення координат. Це дає змогу оптимізувати використання апаратних ресурсів. Для цієї властивості існує перелік шести значень: від найнижчої до найвищої точності.

Друга властивість — distanceFilter. Вона визначає мінімальну відстань у метрах, на яку повинен перемістится пристрій по горизонталі, перш ніж локація оновиться. Таким чином можливо уникнути постійного потоку даних, коли

користувач майже не рухається. Можливими значеннями є будь-які невід'ємні значення в метрах.

У дослідженні [9] протестовано всі можливі конфігурації двох вище наведених властивостей. Найбільш енергоефективною виявилася конфігурація під кодуванням S12: CLLocationAccuracyKilometer для desiredAccuracy та 2^8 метрів для distanceFilter. Найбільш енергозатратними є наступні конфігурації:

- S2 — kCLLocationAccuracyBest для desiredAccuracy та distance = 2^8 метри (середня енергія = 463 Дж);
- S17 — kCLLocationAccuracyBestForNavigation точність і distance 2^8 метри (середня енергія = 463 Дж);
- S5 (за замовчуванням) — kCLLocationAccuracyBest точність та distance 2^4 метри (середня енергія = 460 Дж).

Оскільки стандартна конфігурація входить до трійки найбільш енергомістких, виникає нагальна потреба налаштовувати її власноруч.

2.1.3. Використання локації без потреби

Відповідно до Рисунка 2.2, збільшивши інтервал оновлення з 15 секунд до 2 хвилин, додаток може заощадити 9% заряду батареї за годину [8].

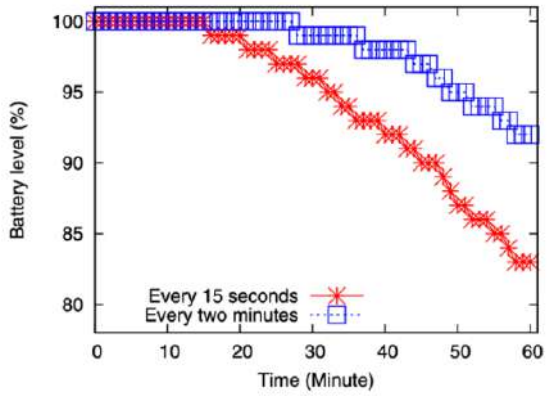


Рис. 2.2. Різниця енергоспоживання з різною частотою запитів локації.

Згідно з зображенимина Рисунку 2.3, використання GPS скоротило час роботи зі 144 годин до 11 годин 40 хвилин, тобто більше ніж у 12 разів [10]. Саме тому важливо вчасно зупиняти оновлення місцеперебування.

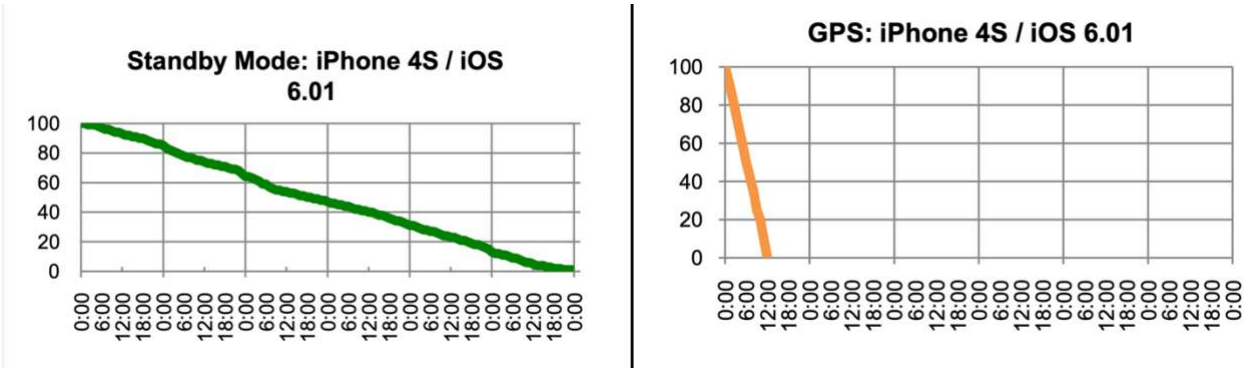


Рис. 2.3. Різниця енергоспоживання з ввімкненими та вимкненими службами геолокації.

Для кожного сервісу LocationManager має окремий метод запуску відстежування локації: startUpdatingLocation(), startMonitoringSignificantLocationChanges(), startMonitoringVisits(). Відповідно, методів для зупинки оновлення локації також є три: stopUpdatingLocation(), stopMonitoringSignificantLocationChanges(), stopMonitoringVisits().

2.2 Використання таймера

2.2.1 Визначення та загальний опис

Таймер — це механізм, який дає змогу запланувати відкладене або періодичне виконання певного коду. Він чекає, доки мине певний проміжок часу, а потім спрацьовує, виконуючи певну дію, наприклад, надсилаючи повідомлення цільовому об'єкту.

Активність таймера тісно пов’язана з енергоспоживанням пристрою. Виведення системи зі стану очікування вимагає витрат енергії, коли процесор та інші системи виводяться з режиму очікування з низьким енергоспоживанням. Якщо таймер змушує систему прокидатися, він несе ці витрати [6].

У Swift найчастіше використовуваними класами для створення таймерів є Timer із фреймворку Foundation та Timer із фреймворку Combine. Обидва підходи надають гнучкі можливості для планування дій.

Створення таймера з Foundation можливо кількома способами:

1. За допомогою використання статичних методів scheduledTimer. Ці методи автоматично додають таймер до поточного циклу обробки подій, що робить їх досить простими у використанні.
2. За допомогою використання ініціалізаторів. Цей підхід надає більше контролю над налаштуванням властивостей таймера перед його запуском власноруч.

Ознайомитися з повним переліком можливих методів та ініціалізаторів можливо в джерелі [11].

Аби створити таймер з Combine, необхідно викликати статичний метод publish(every:tolerance:on:in:options:). Цей метод повертає Publisher, на який необхідно підписатися для початку роботи таймера. Результатом підписки є об’єкт типу AnyCancellable, який і є представленням підписки.

2.2.2 Невчасне вимкнення періодичного таймера

Відсутність вчасного вимкнення таймера, коли потреби в ньому більше немає, призводить до зайвих витрат енергії.

Для зупинки таймера, створеного за допомогою Foundation, потрібно викликати метод invalidate() на його екземплярі. Цей метод також видаляє його з RunLoop, звільнюючи пов’язані з ним системні ресурси.

Аби зупинити таймер з Combine, потрібно викликати метод cancel() в об’єкта AnyCancellable, який став результатом підписки на Publisher таймера.

Якщо знехтувати цією рекомендацією, періодичний таймер продовжить спрацьовувати через заданий інтервал. Відповідно, блок коду, прив'язаний до цього таймера, продовжуватиме викликатися навіть якщо його результат буде більше не потрібним. Він також використовуватиме системні ресурси, що буде негативно впливати на продуктивність програми та час роботи акумулятора на мобільному пристрої.

2.2.3 Відсутність допустимого відхилення часу спрацювання таймера

У контексті ефективного використання таймерів існує поняття Tolerance — максимальна величина, на яку фактичний час спрацювання таймера може відрізнятися від запланованого інтервалу. Операційні системи оптимізують роботу таймерів, згруповуючи їхнє спрацювання для зменшення енергоспоживання та підвищення загальної продуктивності системи. Після зазначення tolerance, таймер може спрацювати в будь-який час між запланованим часом спрацьовування і запланованим часом спрацьовування, доданим до зазначеного tolerance. Таким чином, таймер у жодному разі не спрацює раніше запланованого часу. Для повторюваних таймерів наступний час сповіщення розраховується від початково запланованого часу, аби майбутні сповіщення відповідали початковому розкладу попри можливі затримки в попередніх спрацюваннях.

У Foundation при створенні таймера за допомогою статичних методів scheduledTimer, tolerance за замовчуванням може мати певне невелике значення, встановлене системою. У разі створення таймера за допомогою ініціалізаторів, Timer має властивість tolerance типу TimeInterval, яку можна встановити після його створення, але до його додавання до RunLoop.

У Combine метод publish() також має опціональний параметр tolerance типу DispatchQueue.SchedulerTimeInterval.Tolerance.

Встановлення такої величини надає операційній системі можливість затримати або прискорити спрацювання таймера в межах цього діапазону. Використання такого підходу значно збільшує час, який процесор проводить в режимі очікування, у той час, як користувачі не помічають жодних змін у швидкості реакції системи. Згідно з рекомендаціями для підвищення енергоефективності коду від Apple [6], загальна рекомендація полягає в тому, щоб встановити допуск щонайменше в десять відсотків від інтервалу для повторюваного таймера. Навіть невелике допустиме відхилення має значний позитивний вплив на енергоспоживання застосунків.

2.3 Робота з Bluetooth

2.3.1 Визначення та загальний опис

Bluetooth — це технологія бездротового зв'язку малого радіуса дії, що використовуються для обміну даними між пристроями. Завдяки їй можливо передавати файли та під’єднуватися до периферійних пристроїв, як-от навушників чи смартгодинників.

Застосунки використовують цю технологію для безперервного сканування на наявність певних пристроїв або підтримки активного з'єднання з ними. Використання енергії зростає зі збільшенням частоти сканування, тривалості активних з'єднань та обсягу переданих даних. Як наслідок, постійне активне сканування Bluetooth може швидко виснажувати акумулятор смартфона.

В iOS робота з пристроями, що підтримують безпровідну технологію Bluetooth low energy (BLE), відбувається за допомогою фреймворку Core Bluetooth. Усі операції, пов’язані з BLE, здійснюються за допомогою екземпляра класу CBCentralManager. Він відповідає за ініціалізацію та керування станом Bluetooth, пошук доступних пристроїв і встановлення та керування підключеннями до знайдених пристроїв.

Відповідно до дослідження [7], а також офіційної інструкції з мінімізації енергоспоживання від Apple [6], однією з найбільш енергомістких операцій, пов’язаних з Bluetooth, є постійний пошук девайсів.

2.3.2 Постійний пошук пристроїв

Навіть якщо пристрій не під'єднаний до будь-якого іншого пристрою поруч, він все ще шукає інші девайси навколо та намагається з'єднатися з ними.

Виявлення віддалених периферійних пристроїв розпочинається після виклику методу scanForPeripherals(withServices:options:) класу CBCentralManeger. Цей метод запускає процес активного радіосканування, під

час якого центральний менеджер Bluetooth пристрою надсилає запити та очікує на рекламні пакети від пристроїв, що знаходяться в радіусі дії.

Коли потреба у виявленні пристроїв більше не є актуальною, необхідно зупинити процес пошук. Для цього використовують метод stopScan() класу CBCentralManager.

2.4 Використання GPU

2.4.1 Визначення та загальний опис

GPU (graphics processing unit) — це особливий тип процесора, який відповідає за обробку графічних завдань [7].

Згідно з офіційною інформацією від Apple, якщо в застосунку є нестандартні вікна та елементи керування, розробники повинні переконатися, що код відмальовування працює ефективно. Застосунок не повинен оновлювати вміст без потреби, наприклад, у затемнених ділянках екрана. Щоразу, коли застосунок оновлює вміст на екрані, він вимагає, щоб CPU, GPU та екран були активними. Зайве або неефективне малювання може виводити системні ресурси зі стану низького енергоспоживання або взагалі перешкоджати їхньому вимкненню, що призводить до значного споживання енергії [6].

Для роботи з графічним інтерфейсом в iOS існує два фреймворки:

1. UIKit — імперативний фреймворк, що має багатий набір готових компонентів та хороші інструменти для обробки подій користувача, анімації, власноруч створених графічних елементів тощо.

2. SwiftUI — сучасний декларативний фреймворк для створення графічного інтерфейсу на всіх платформах Apple. Декларативний синтаксис робить код більш лаконічним та читабельним.

Apple надає наступні рекомендації, пов’язані з роботою з графічним інтерфейсом:

- мінімізувати використання напівпрозорих елементів;
- мінімізувати використання розмитих елементів [6].

Хорошою практикою також є мінімізація використання тіней [12] [13].

2.4.2 Використання напівпрозорих елементів

Apple радить мінімізувати використання непрозорості, оскільки це може спричинити надмірне використання графічного процесора через рендеринг користувацького інтерфейсу. Коли елемент є частково прозорим, системи необхідно об'єднати його змістом, що знаходиться під ним, піксель за пікселями. Цей процес вимагає додаткових обчислювальних ресурсів, особливо для складних ієрархій елементів або анімацій.

Для елемента, створеного за допомогою фреймворку UIKit, відсоток непрозорості налаштовується за допомогою властивості alpha. Ця властивість має тип CGFloat, який може приймати значення в діапазоні від 0.0 до 1.0 включно. Елемент зі значенням alpha = 0.0, буде повністю прозорий, а з 1.0 — непрозорий. Код для налаштування непрозорості на 50% виглядатиме наступним чином: myView.alpha = 0.5.

Для елемента, створеного за допомогою фреймворку SwiftUI, задати рівень непрозорості можливо використовуючи модифікатор opacity(_:). Цей модифікатор приймає аргумент типу Double. Значення задається ідентично до властивості alpha в UIKit. Зміна відсотка непрозорості на 30% матиме такий вигляд: opacity(0.3). Особливо шкідливим для енергії є використання непрозорості для зображення у SwiftUI, адже в такому разі система спочатку відтворює його з повною непрозорістю, а потім накладає прозорий шар [12].

2.4.3 Використання розмитих елементів

Розмиття — це візуальний ефект, який робить зображення або його частину розфокусованим та менш чітким.

Як у SwiftUI, так і в UIKit для розмиття використовується Гауссове згладження [14]. Суть цієї техніки полягає в оновленні значення пікселя шляхом обчислення середнього значення сусідніх пікселів.

В UIKit розмиття створюється за допомогою UIBlurEffect(style:), який передають в UIVisualEffectView(effect:), який потім додають до певного view за допомогою addSubview(_:). Під час його презентації Apple зазначили, що він є досить витратним [15].

UIBlurEffect має три стилі: Extra light, Light та Dark. Приклад того, як вони змінюють користувацький інтерфейс, наведено на Рисунку 2.4.

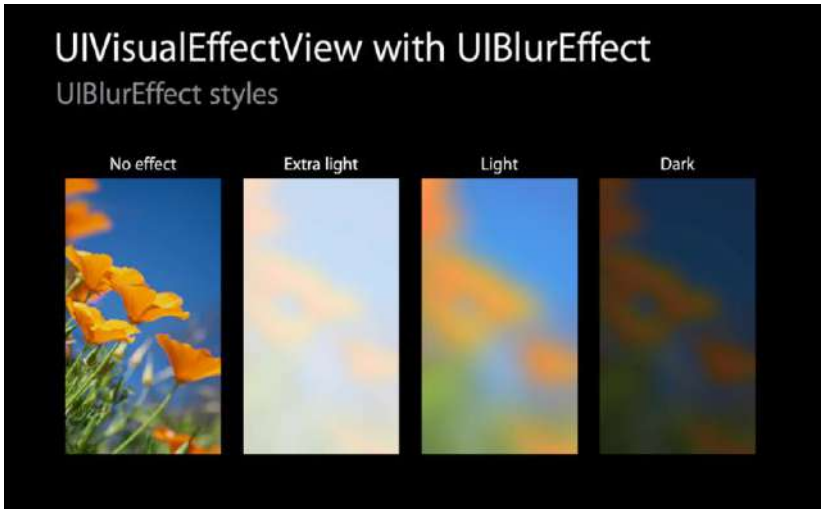


Рис. 2.4. Приклади використання стилів UIBlurEffect.

Як помітно на Рисунку 2.5, найбільше системних ресурсів споживає UIBlurEffectStyleExtraLight

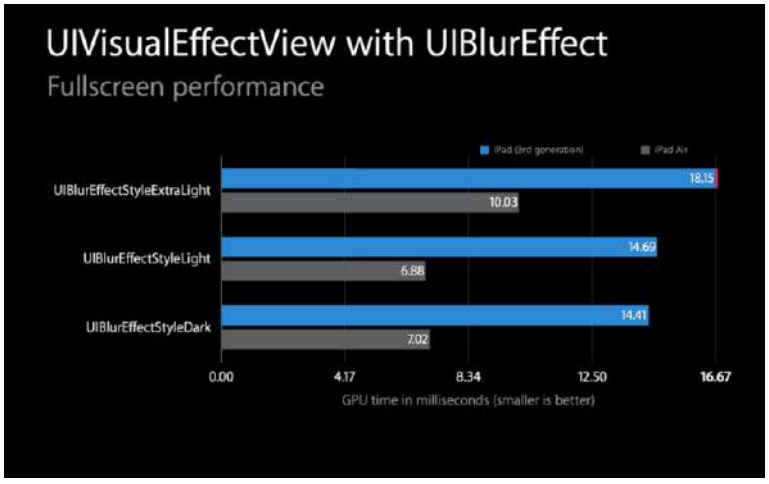


Рис. 2.5. Порівняння продуктивності стилів UIBlurEffect.

Що стосується SwiftUI, створити ефект розмиття можна значно легше: для цього існує модифікатор blur(radius:opaque:), де radius відповідає за інтенсивність розмиття (що більшим є це значення, то розмитішим буде елемент), а opaque — булеве значення, яке вказує на те, чи будуть розмиті також краї графічного елемента.

У статті [16] досліджено вплив використання напівпрозорости, поєднаної з розмиттям, на споживання енергії. На Рисунку 2.6 наведено порівняння використання системних ресурсів з увімкненими (+T) та вимкненими (-T) напівпрозорістю та розмиттям. На основі отриманих даних автор зробив висновок, що використання вище згаданих графічних ефектів збільшило навантаження на GPU, CPU та DRAM. Це призвело до збільшення загальної потужності корпусу на ~200 мВт. Збільшення енергоспоживання на 200 мВт спричинить зменшення тривалості роботи від акумулятора 14-дюймового MacBook Pro на 20 хвилин.

Ці результати мають також важливе значення для мобільних пристроїв на iOS, де аналогічні графічні ефекти реалізуються за допомогою GPU. Через менший об’єм акумулятора в iPhone, додаткове навантаження на GPU впливатиме на тривалість автономної роботи ще суттєвіше. Хоча дослідження проводилося на Macbook, отримані результати є релевантними для iOS-пристроїв. Механізм обчислення розмиття — частина рендерингу, що спільно використовується в усій екосистемі Apple.

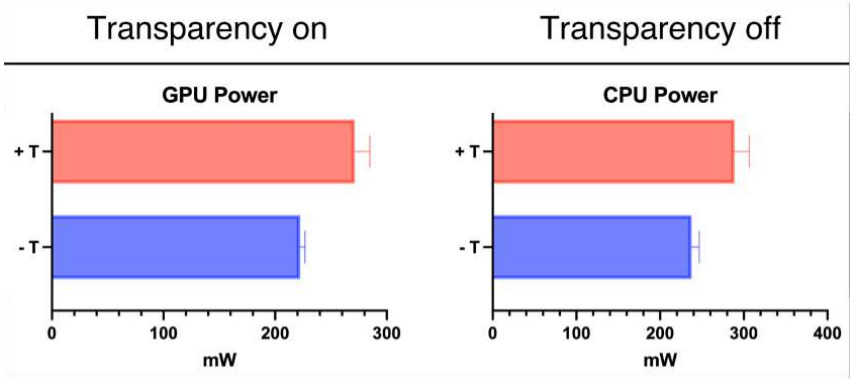


Рис. 2.6. Порівняння споживаної енергії з ввімкненими та вимкненими напівпрозорістю та розмиттям.

1.4.4 Використання тіней

Shadow дає змогу додати тінь позаду графічних елементів. Для неї можна налаштовувати колір і відступ у горизонтальному та вертикальному вимірах, а також розмивати краї тіні. Останнє може спричинити особливо надмірне енергоспоживання, оскільки поєднує в собі напівпрозорість та розмиття. Використання тіні може негативно впливати на енергоспоживання, особливо у великих View чи анімаціях [12].

У SwiftUI для створення тіні необхідно викликати метод shadow(color:radius:x:y:).

В UIKit синтаксис є більш комплексним, адже для кожної дії існує окрема властивість, якій можна задати значення за потреби. Тому створення тіні виглядатиме приблизно наступним чином:

- myView.layer.shadowColor = UIColor.black.cgColor
- myView.layer.shadowOpacity = 1
- myView.layer.shadowOffset = .zero
- myView.layer.shadowRadius = 10

Найважливішою властивість тут є shadowOpacity, оскільки якщо вона рівна нулю, тінь не буде видно, відповідно, ресурси на неї не витрачатимуться.

Існує можливість закешувати тінь, яку система вже зарендерила, аби не поставала потреба в повторному її відмальовуванні. Досягти цього можна за допомогою встановлення значення true для yourView.layer.shouldRasterize, після чого варто встановити myView.layer.rasterizationScale = UIScreen.main.scale, аби тінь кешувалася в тому ж розмірі, як і головний екран та не була піксельною.

2.5 Використання CPU

2.5.1 Визначення та загальний опис

CPU — основний процесор комп’ютера. Згідно з кількома дослідженнями [7][17], CPU є найбільшим споживачем енергії серед усіх, що відображено на Рисунку 2.7.

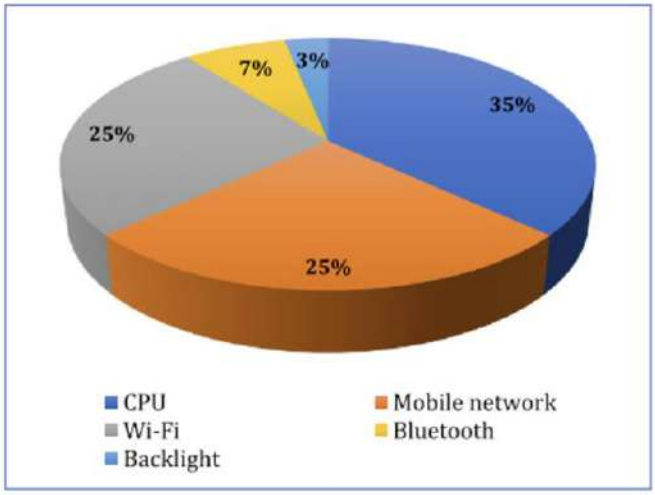


Рис. 2.7. Розподіл використання енергії різними компонентами [7].

Нещодавні дослідження доводять, що неефективні рішення під час розробки мають негативний вплив на енергоспоживання мобільних застосунків. До прикладу, роботу [4] присвячено дослідженню того, як ті чи інші патерни дизайну впливають на енергоспоживання. Згідно з джерелом [7], застосунки можна оптимізувати для того, щоб ефективніше виконувати операції, що суттєво зменшує енергоспоживання центрального процесора та, відповідно, зменшує споживання енергії.

2.5.2 Нестатичні функції, які не звертаються до властивостей класу

Однією із поширених енергомістких практик є наявність нестатичних методів всередині класу, які могли б бути статичними, оскільки не використовують жодні поля класу чи структури, не викликаються за допомогою інших нестатичних методів та самі не викликають їх. Такий тип функцій має назву MIM, що є абревіатурою для member ignoring method.

Способом рефакторингу таких шматків коду є впровадження статичного метода. Як виявилося, таке покращення суттєво впливає на енергоефективність мобільного застосунку. У середньому енергоспоживання методів змінюється від

0,070 Дж до 0,008 Дж, що призводить до визначення методу в дев'ять разів ефективнішого [18]. У роботі [20] зазначено, що виправлення цього антипатерну сприяло зниженню енергоспоживання застосунку на 15%.

На рисунку 2.8 наведено різницю в енергоспоживанні між початковим кодом, що містить такий антипатерн (MIM), та оптимізованим кодом, у якому цю шкідливу практику було виправлено (R-MIM) [18].

Smell type	Min	1st Qu.	Median	Mean	3st Qu.	Max
MIM	0.001	0.002	0.004	0.04	0,028	0.981
R-MIM	0.0001	0.002	0.018	0.02	0.019	0,038

Рис. 2.8. Різниця між споживанням енергії

Дані щодо впливу енергомісткого коду на тривалість роботи батареї зображено на рисунку 2.9.

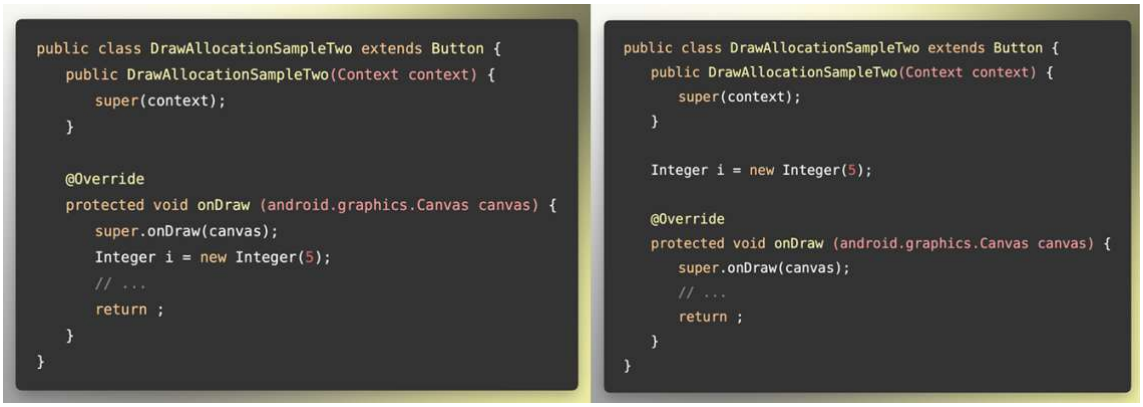
Smell type	% of Battery Discharge Before	% of Battery Discharge After
MIM	$1.4 * 10^{-6}$ %	$6.2 * 10^{-7}$ %

Рис. 2.9. Різниця у швидкості розряду акумулятора

Відповідно до джерела [19], статичні методи зберігаються у блоці пам'яті, відокремленому від блоку, де зберігаються об'єкти. Незалежно від того, скільки екземплярів класу буде створено протягом часу виконання програми, лише один екземпляр такого методу буде створений. Тож цей механізм сприяє зменшенню споживання енергії.

2.5.3 Draw Allocation

Draw Allocation — ситуація, за якої нові об’єкти створюються безпосередньо з операціями малювання чи розмітки інтерфейсу, які є дуже чутливими до продуктивности. Іншими словами створення об’єктів усередині функції onDraw() (Java) є шкідливою практикою [19][21][22]. Прикладом такої шкідливої практики може бути ініціалізація будь-яких об’єктів. На рисунку 2.10 зображено рефакторинг створення цілого числа в межах методу onDraw().



```
public class DrawAllocationSampleTwo extends Button {
    public DrawAllocationSampleTwo(Context context) {
        super(context);
    }

    @Override
    protected void onDraw (android.graphics.Canvas canvas) {
        super.onDraw(canvas);
        Integer i = new Integer(5);
        // ...
        return ;
    }
}
```

```
public class DrawAllocationSampleTwo extends Button {
    public DrawAllocationSampleTwo(Context context) {
        super(context);
    }

    Integer i = new Integer(5);

    @Override
    protected void onDraw (android.graphics.Canvas canvas) {
        super.onDraw(canvas);
        // ...
        return ;
    }
}
```

Рис. 2.10. Приклад енергомісткого та енергоефективного коду [22].

Рекомендованим рішення є винесення створення об'єктів за межі методу малювання, як показано на Рисунку 2.12, де ініціалізація цілого числа відбувається поза відмальовуванням графічних елементів. У такий спосіб, у методі onDraw() Integer повторно використовується для кожної операції малювання.

У попередньому фрагменті коду новий екземпляр класу Integer створювався при кожному виконанні функції onDraw(). Після рефакторингу коду створення об’єкта вилучено з операції малювання, завдяки чому виконується виключно один раз під час виконання програми.

В одному з досліджень [21] виправлення цієї шкідливої практики в одному з тестових застосунків призвело до економії енергоспоживання на 1.47%, що дорівнює 21 хвилині роботи мобільного пристрою, як вказано на Рисунку 2.11.

Application	Pattern	MD	Cohen's d	IMP (%)	Savings (min)
Talarmo	DrawAllocation	-0.86	-1.11	1.47	21

Рис. 2.11. Дані про зниження енергоспоживання після рефакторингу[21]

В операційній системі iOS схожим за функціональністю є метод екземпляра класу UIView draw(_ rect: CGRect), що використовується для малювання об’єктів власноруч у межах UIView. Приклади подібного енергомісткого та енергоефективного коду, написаного мовою Swift, зображено на Рисунку 2.12.

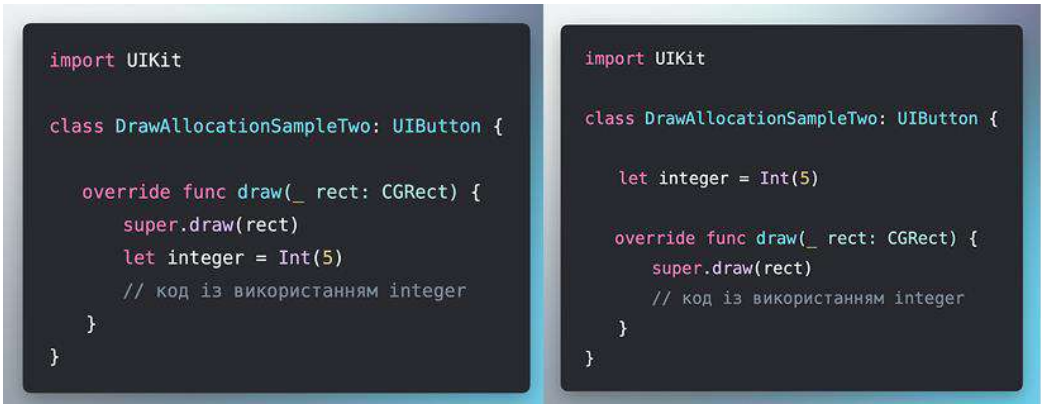


Рис. 2.12. Енергомісткий та енергоефективний код

Антипатерн Draw Allocation є типовим прикладом того, як навіть дрібні недоліки в архітектурі застосунку можуть мати вплив на енергоспоживання. Його виправлення переважно не вимагає значних змін у логіці застосунку, проте позитивно впливає як на продуктивність, так і на енергоспоживання мобільного пристрою.

2.5.5 Inline method

Вбудовування методу — назва ситуації, в якій тіло певного методу складається виключно з простого делегування задачі іншому методу. У дослідженні [24] виправлення цього антипатерну сприяло зниженню енергоспоживання з 44.059 до 43.98 мікроджоулів, що складає 0.2%.

Виправлення цієї проблеми є досить простим — перенести код із внутрішньої функції до зовнішньої.

2.5.6 Методи без параметризованого об'єкта

Коли у функцію передано три чи більше параметрів, варто спробувати замінити їх одним об'єктом, що міститиме ці параметри як власні поля. До прикладу, замість того, аби передавати до метода окремими об'єктами ім'я, прізвище, по-батькові, номер телефону тощо, варто зробити окремий об'єкт Людина.

У дослідженні [24] завдяки введенню такого об'єкта енергоспоживання знизилося на 6%: з 45.029 мікроджоулів до 44.771 мікроджоулів.

2.6 Висновки до розділу

У цьому розділі оглянуто найпоширеніші антипатерни, що негативно впливають на енергоспоживання мобільних застосунків. Для кожного з них надано типові приклади та рекомендований спосіб оптимізації.

Більшість енергомістких практик є релевантними як для iOS, так і для Android, оскільки реалізуються за допомогою схожих API. Виявлені антипатерни мають доведений безпосередній вплив на тривалість автономної роботи пристроїв, що підтверджено у відповідних дослідженнях. Наведені приклади демонструють, що навіть невеликі зміни в коді можуть знизити енергоспоживання. Описані антипатерни базуються на результатах наукових досліджень та офіційних порад від Apple.

Через чималу кількість можливих джерел витоку енергії важливим є створення інструменту для автоматичного виявлення таких шкідливих практик на етапі компіляції. Це стало темою наступного розділу.

РОЗДІЛ 3.
РОЗРОБКА СТАТИЧНОГО АНАЛІЗАТОРА ЕНЕРГОСПОЖИВАННЯ

У цьому розділі описано розробку інструменту для статичного аналізу коду мовою Swift під назвою EnergyUsageAnalyzer. З ним можливо працювати двома способами: за допомогою командного рядка або ж інтегрувавши його безпосередньо у проєкт у середовищі розробки XCode.

Зручнішим варіантом є інтеграція в XCode, оскільки вона дає змогу виконувати аналіз одразу всіх файлів проєкту під час збірки без необхідності запускати інструмент вручну для кожного файлу. Водночас реалізовано можливість пофайлового запуску з термінала для швидкої перевірки окремих файлів або ж під час налагодження інструмента при доповненні його власними правилами.

3.1 Імплементація аналізатора

3.1.1 Налаштування пакета

Для реалізації інструменту було обрано формат Swift Package, який є стандартним способом поширення коду. За допомогою нього створюється код, який можна перевикористати, організувати, поширити між розробниками та інтегрувати в проєкти.

Першим кроком стало створення пакета за допомогою введення команди swift package init --type executable в терміналі. Після цього налашовано Package.swift — маніфест SPM, що описує Swift-пакет, який має CLI-інструмент та build tool plugin.

Наступне налаштування — опис залежностей. Пакет використовує дві залежності: swift-argument-parser для парсингу аргументів та swift-syntax для аналізу коду.

Опісля, як зображено на Рисунку 3.1, описано targets: .executableTarget для створення виконуваного файлу та .plugin для створення плагіну, що використовуватиметься під час build phase.

```
27      targets: [
28          .executableTarget(
29              name: "EnergyUsageAnalyzer",
30              dependencies: [
31                  .product(name: "SwiftSyntax", package: "swift-syntax"),
32                  .product(name: "SwiftParser", package: "swift-syntax"),
33                  .product(name: "ArgumentParser", package: "swift-argument-parser"),
34                  .product(name: "Yams", package: "Yams"),
35              ]
36          ),
37          .plugin(
38              name: "EnergyUsageAnalyzerPlugin",
39              capability: .buildTool(),
40              dependencies: [
41                  .target(name: "EnergyUsageAnalyzer")
42              ]
43          )
44      ]
```

Рис. 3.1. Опис targets

Наступний крок — реалізація плагіна, що має тип BuilToolPlagin, який дає змогу запускати зовнішній інструмент під час компіляції. Він автоматично запускатиметься для кожного Swift-файл, аби аналізувати код на наявність неефективного енергоспоживання.

Для цього створено файл plugin.swift, який складається з двох частин: створення основної структури плагіна та підтримка XCode-проєктів.

Основна структура має назву EnergyUsageAnalyzerPlugin та має атрибут @main, аби означити точку входу в плагін. Аби структура відповідала протоколу BuildToolPlugin, потрібно створити метод createBuildCommands(context: PluginContext, target: Target), що її викликає Swift Package Manager під час побудови пакета. Ця функція створює список команд для запуску аналізатора під час збірки.

У наступній частині коду відбувається перевірка можливості імпорту модуля XCodeProjectPlugin, аби забезпечити сумісність плагіна з XCode-проєктами. Якщо модуль доступний, ми розширюємо раніше створену структуру, додаючи відповідність протоколу XcodeBuildToolPlugin. До цього реалізовано метод із сигнатурою, ідентичною до вище зазначеного, у якому

фільтруються всі Swift-файли та створюється команда запуску аналізатора для кожного з них.

Для забезпечення гнучкості налаштувань та адаптації інструмента до потреб розробника додано можливість вимкнення окремих правил аналізатор. Цей функціонал реалізовано за допомогою YAML-файлу під назвою energy-analyzer.yml. Перший його рядок визначає ключ disabled_rule, після якого потрібно перерахувати правила, які потрібно деактивувати. Назву кожного правила потрібно написати з нового рядка, поставивши перед ним дефіс.

У такий спосіб, реалізовано базову архітектуру Swift-пакета, налаштовано його конфігурацію та закладено первинну логіку, необхідну для подальшої імплементації аналізатора.

3.1.3 Створення основного файлу

У файлі XcodeAnalyzer.swift реалізовано логіку роботи з параметрами, початком аналізу та виведення отриманих даних у консоль чи як попередження в XCode. Для роботи з вхідним параметром застосовано бібліотеку ArgumentParser, яка дає змогу створювати інтерфейс командного рядка з аргументами.

Обов’язковим аргументом під назвою inputFilePath повинен бути шлях до Swift-файлу, який потрібно проаналізувати. Цей шлях зчитується за допомогою @Option.

Другий аргумент, configPath, є опційним. У нього передається шлях до власного файлу з деактивованими правилами. Якщо цей параметр не задано, програма звертається до власного файлу, що знаходить в кореневій папці проєкту.

Головною точкою запуску, що виконується при виклику утиліти, є функція run(). Після введення команди в ній відбувається перевірка на те, чи файл за введеним шляхом існує та відбувається читання вмісту файлу за допомогою

SwiftParser. Після цього отриманий файл, що містить абстрактне синтаксичне дерево, передано на подальший аналіз. Основна логіка знаходиться в методі analyzed(file:), який викликає почергово всі координатори. Після проходження всіма координаторами файлом, результати аналізу об'єднуються в один масив повідомлень типу WarningMessage, що виводяться у форматованому вигляді в термінал або відображаються як попередження в XCode.

Структуру WarningMessage, що містить як поля шлях до файлу, line, column та message, створено для того, аби XCode міг коректно відображати попередження в потрібному місці в коді.

3.1.4 Створення відвідувачів

Логіку пошуку антипатернів поділено на відвідувачів та координаторів. Усі відвідувачі є нащадками класу SyntaxVisitor з бібліотеки SwiftSyntax. При ініціалізації усім відвідувачам передається шлях до файлу, аналіз якого здійснюється. Це потрібно для того, аби відвідувач мав змогу отримати інформацію про line та column коду, де знайдено шкідливу практику.

Кожен відвідувач реалізовує один метод visit(), окрім InstanceFunctionsVisitor та BlurPropertyVisitor.

InstanceFunctionsVisitor містить два методи, що реалізують однаковий алгоритм. Відмінність полягає в тому, що один із методів працює з класами, а інший — зі структурами.

```
func search(_ members: MemberBlockItemListSyntax, _ location: SourceLocation) {
    for member in members {
        if let funcDecl = member.decl.as(FunctionDeclSyntax.self) {
            if let bodyContent = funcDecl.body?.description,
               funcDecl.modifiers.isEmpty {
                let warningMessage = WarningMessage(filePath: filePath, line: location.line, column:
                    location.column, message: "Found functions that can be static (instance_identifier_rule)")
                codeBlocks.append((bodyContent.trimmingCharacters(in: .whitespacesAndNewlines),
                    funcDecl.description.trimmingCharacters(in: .whitespacesAndNewlines), warningMessage))
            }
        }
    }
}
```

BlurPropertyVisitor має дещо складнішу логіку, оскільки працює з двома типами виразів:

- оголошення змінних — ініціалізація UIBlurEffect та UIVisualEffectView;
- виклик функції — використання метода addSubview(), де параметром є знайдений раніше UIVisualEffectView.

У випадку, якщо код задовільнив кілька умов, як-от наявність виклику певного метода чи відсутність аргумента з певною назвою, записується відповідне попередження типу WarningMessage.

Після проходження усього файла та знаходження антипатернів відвідувачем, отримані дані передаються координаторові, який опрацьовує їх та повертає в основний файл, звідки було викликано функцію аналізу.

3.1.5 Створення координаторів

Для кожної шкідливої практики реалізований проєкт має окремі координатори. Кожен з них використовує від одного до трьох відвідувачів. Усі координатори відповідають протоколу EnergyCoordinator, що вимагає наявність методу analyze(_ sourceFile: SourceFileSyntax). Для зручності запуску лінтера а в структурі Const є функція getAllCoordinators(disabled:), яка повертає масив відфільтрованих координаторів. Фільтрація необхідна для того, аби не відбувався пошук порушення правил, які було деактивовано.

У методі analyze відбувається ініціалізація усіх необхідних відвідувачів для виявлення певного антипатерна. Відвідувачі проходять по файлу після виклику методу walk(_ node: some SyntaxProtocol).

Подальші кроки залежать від складності знаходження антипатерну та необхідних дій над знайденими елементами. Існує два сценарії:

1. Для координаторів з одним відвідувачем. У такому випадку координатор повертає масив знайдених порушень правил. Прикладами таких координаторів є DrawAllocationCoordinator та ParameterObjectCoordinator.
2. Для координаторів із кількома відвідувачами. Вони містять додаткову логіку оброблення отриманих даних.

Як приклад другого сценарія розгляньмо клас BluetoothCoordinator. Він отримує дані від двох відвідувачів — BluetoothStartVisitor та BluetoothStopVisitor.

Перший клас знаходить та зберігає усі імена об’єктів, які викликають метод scanForPeripherals(), а також записує попередження, що містять інформацію про знайдений антипатерн.

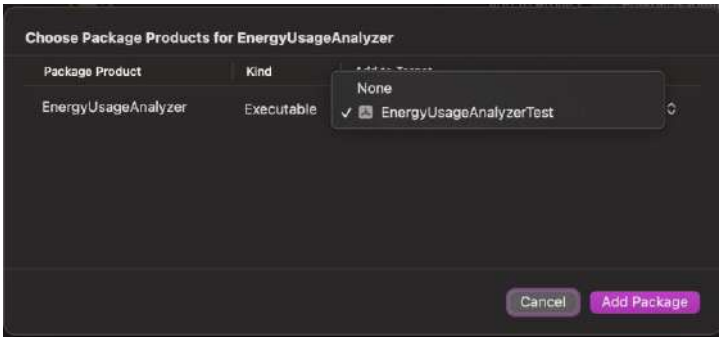
Другий відвідувач шукає імена об’єктів, які викликають stopScan(). У цьому випадку помилку виявити просто — вона полягатиме у початку сканування та відсутності його зупинки. Після отримання масивів з обох відвідувачів, масив повідомлень першого відвідувача фільтрується за фактом наявності використання якогось зі знайдених об’єктів.

Такий підхід до організації координаторів забезпечує масштабованість аналізатора: правила пошуку антипатернів можливо додавати без зміни загальної архітектури. Завдяки протоколу EnergyCoordinator знаходження кожної шкідливої практики залишається зручним для тестування та підтримки.

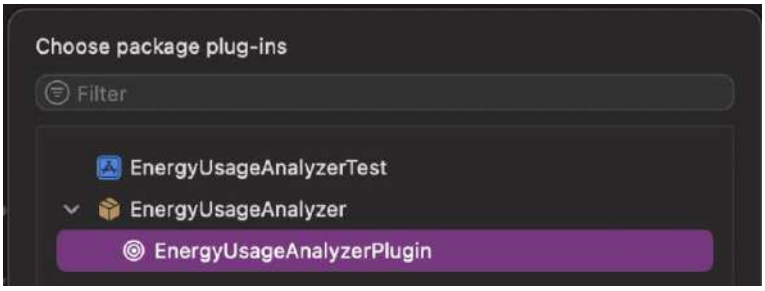
3.2 Інтеграція плагіна в середовище розробки XCode

3.2.1 Додавання плагіна до проєкту

Додавання плагіна відбувається у два прості кроки. Перший — додати пакет до залежностей проєкту. Для цього потрібно перейти у Project -> Package Dependencies та натиснути на символ «+», після чого у поле пошуку ввести покликання на GitHub-репозиторій [26], обрати необхідний target та натиснути на Add Package. Додана залежність з’явиться у списку.



Другий крок — додати build tool plug-in. Спершу необхідно перейти у Targets -> Build Phases. Опісля розгорнути вкладку Run Build Tool Plug-ins, натиснути на символ «+» та обрати плагін під назвою EnerguUsageAnalyzerPlugin.



Аби деактивувати певнів правила, необхідно створити файл файл "energy-analyzer.yml" у кореневій папці проєкту та внести в нього перелік правил.

Після цього можна запускати збірку проєкту, аби відбувся аналіз коду на наявність енергомістких практик.

3.2.2 Приклад роботи плагіна

Розгляньмо роботу лінтера на прикладі Swift-файлу, в якому відбувається робота з таймером. У коді наявні два порушення правил: відсутність визначення можливої похибки таймера та відсутність його зупинки. На Рисунку 3.2 помітно, що аналізатор вдало впорався з поставленими задачами. Варто зазначити, що в цьому випадку попередження відображаються в тому рядку, де таймер ініціалізовано. Це зроблено для скорочення часу розробника, який витрачається на пошук місця помилки.

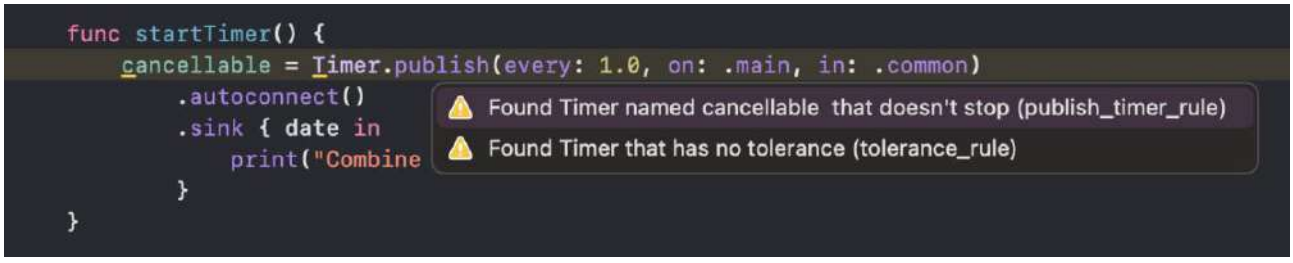


Рис. 3.2. Демонстрація роботи аналізатора в XCode

Розгляньмо приклад, в якому порушено одразу три правила роботи з GPU: використання тіні, прозорості та розмиття. Як зображено на Рисунку 3.3, усі

помилки відображаються на об’єкті, до якого було застосовано той чи інший модифікатор, що спричиняє надмірне споживання енергії.

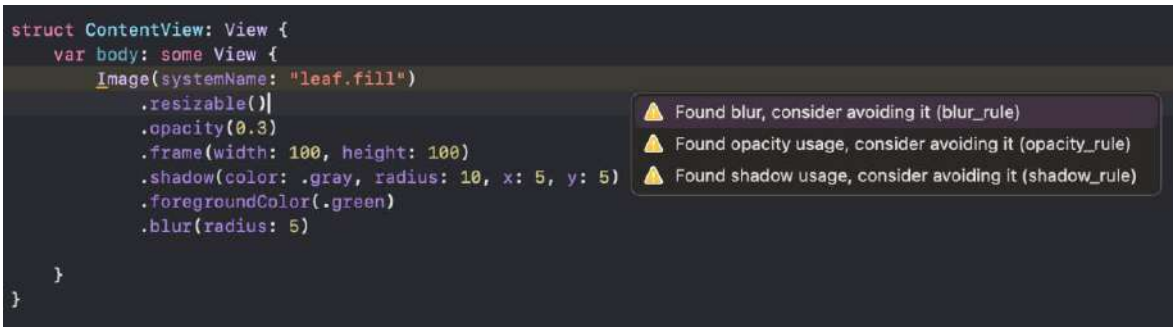


Рис. 3.3. Демонстрація роботи аналізатора в XCode

Отже, аналізатор коректно працює та вказує на енергомісткі практики коду, забезпечуючи зручний інтерфейс та детальні повідомлення з порадами того, як покращити написаний код.

3.3 Робота з інструментом за допомогою командного рядка

3.3.1 Налаштування та запуск аналізатора

Інструмент також можливо запускати безпосередньо з командного рядка. Для цього необхідно завантажити Swift-пакет з GitHub-репозиторію [26] та перейти через термінал у його директорію. Для запуску аналізатора потрібно ввести команду "swift run EnergyUsageAnalyzer -i <шлях до файлу відносно проєкту>".

Аби додати власний yml-файл з переліком правил, що потрібно деактивувати, до команди потрібно додати "-c <шлях до файлу відносно проєкту>". Якщо цей аргумент не задано, використовується файл під назвою "energy-analyzer.yml" у кореневій папці проєкту, який можливо змінювати за потреби.

3.3.2 Приклад роботи аналізатора

Звіт з аналізом містить лише знайдені антипатерни, таким чином можливо уникнути зайвої інформації та дати змогу розробнику зосередитися на потенційно проблемних місцях у коді.

На Рисунку 3.4 зображено результат роботи аналізу для коду, що містить кілька типових проблем, пов’язаних із неефективним використанням CPU:

- використання вбудованого метода;
- нестатичні функції, які не звертаються до властивостей класу;
- методи без параметризованого об’єкта

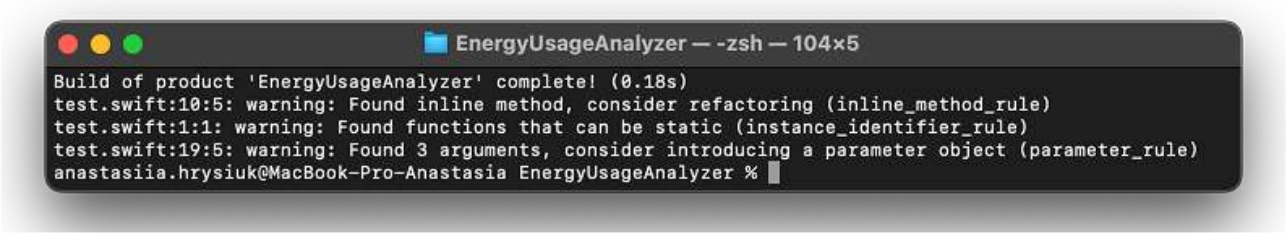


Рис. 3.4. Звіт аналізатора

У створеному звіті користувач може переглядати назву файлу, номер рядка та символа, де було знайдено помилку, а також повідомлення про знайдену помилку та пораду щодо її рефакторингу. Це полегшує пошук розташування помилки в коді та сприяє розумінню того, що саме в написаному коді шкодить акумуляторові.

3.4 Висновки до розділу

У цьому розділі описано покрокове створення статичного аналізатора EnergyUsageAnalyzer, налаштовуючи його як для роботи за допомогою командного рядка, так і для відображення попереджень у XCode. Детально оглянуто процес налаштування пакету, що стане в пригоді розробникам, що прагнуть створити власний аналізатор та інтегрувати його в середовище розробки XCode.

Особливо увагу приділено деталям роботи відвідувачів та координаторам. Для ширшого розуміння деталей реалізації наведено приклади того, як працюють конкретні класи.

Також було проведено тестування створеного лінтера на прикладі коду мовою Swifty, що містять поширені антипатерни. Інструмент успішно виявив випадки неефективного використання різних API: використання енергомістких модифікаторів, відсутність зупинки сканування Bluetooth тощо.

Результати тестування підтвердили здатність аналізатора працювати на етапі збірки та без потреби в запуску коду. У такий спосіб, тестування підтвердило працездатність та практичну доцільність використання такого лінтера для коду мовою програмування Swift.

ВИСНОВКИ

У результаті цієї роботи було створено інструмент для статичного аналізу коду мовою програмування Swift. У роботі розглянуто особивості аналізу коду за допомогою бібліотеки SwiftSyntax. Оглянуто наявні інструменти для аналізу коду мобільних застосунків на операційні системі Android. Проаналізовано десятки наявних робіт у цій сфері, завдяки чому вдалося сформулювати перелік шкідливих практик коду за п'ятьма категоріями: робота з локацією, робота з Bluetooth, робота з таймером, використання CPU та використання GPU. Для кожного такого антипатерна описано приклади неефективного коду та поради для його рефакторингу.

Створений аналізатор стане в пригоді тим, хто прагне дбати про енергоефективність своїх застосунків, надавши автоматичне виявлення невдалих рішень при написанні програмного коду. Аналізатор має формат Swift-пакета, що дає змогу легко інтегрувати його у власний проєкт у середовищі розробки XCode, а також працювати з ним за допомогою командного рядка. Використовуючи аналізатор, створений у рамках цієї кваліфікаційної роботи, можна суттєво скоротити час, що витрачається на пошук надмірного енергоспоживання власноруч, зокрема при написанні проєктів, що містять чималу кількість Swift-файлів. Завдяки поділу класів на відвідувачі та координатори можливо додавати власні правила перевірки коду та тестувати кожне з них окремо, не змінюючи при цьому архітектури застосунку.

З огляду на зростання уваги до енергоефективності мобільних застосунків, створений аналізатор має потенціал для подальшого розвитку — зокрема, бути доповненим новими правилами для підтримки сучасних підходів до написання коду та змін у мові програмування Swift.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ferreira D., Anind K. Dey, Kostakos V. Understanding human–smartphone concerns: A study of battery life. In Proceedings of the 9th International Conference on Pervasive Computing, 2011. С. 19–33. URL: https://link.springer.com/chapter/10.1007/978-3-642-21726-5_2 (дата звернення: 25.05.2025)
2. Guihot H. Pro Android Apps Performance Optimization. URL: https://thuvienso.dau.edu.vn:88/bitstream/DHKTDN/6980/1/6278.Pro%20Android%20apps%20performance%20optimization.pdf (дата звернення: 25.05.2025)
3. Pinto G., Castor F. Energy efficiency: A new concern for application software developers. Communications of the ACM, Volume 60, Issue 12, 2017. С. 68-75. URL: https://doi.org/10.1145/3154384 (дата звернення: 25.05.2025)
4. Sahin C., Cayci F., Guti´errez I. L. M., Clause J., Kiamilev F., Pollock L., Winbladh K. Initial explorations on design pattern energy usage. 2012 First International Workshop on Green and Sustainable Software (GREENS), 2012. С. 55–61. URL: https://doi.org/10.1109/GREENS.2012.6224257 (дата звернення: 25.05.2025)
5. Mahmudowa S. J. GREEN CODING IN PROGRAMMING AND PRACTICES. Journal of Engineering and Technology (JET), 15(2), 2025. С. 1-17. URL: https://doi.org/10.54554/jet.2024.15.2.018 (дата звернення: 25.05.2025)
6. Apple Inc. Energy Efficiency Guide for iOS Apps [Електронний ресурс] / Apple Inc. URL: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/index.html (дата звернення: 25.05.2025)
7. P. K. D. Pramanik et al., Power Consumption Analysis, Measurement, Management, and Issues: A State-of-the-Art Review of Smartphone Battery and Energy Usage. IEEE Access, vol. 7, 2019. С. 182113-182172. URL: 10.1109/ACCESS.2019.2958684 (дата звернення: 25.05.2025)

8. Zhuang Z., Kim K.-H., Singh, J. P. Improving energy efficiency of location sensing on smartphones. MobiSys '10: Proceedings of the 8th international conference on Mobile systems, applications, and services, 2010. С. 315-330. URL: https://doi.org/10.1145/1814433.1814464 (дата звернення: 25.05.2025)
9. Bangash A. A., Tiganov D., Ali K., Hindle A. Energy Efficient Guidelines for iOS Core Location Framework. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 2021. С. 320-331. URL: 10.1109/ICSME52107.2021.00035 (дата звернення: 25.05.2025)
10. Narzt W. Power-saving localization techniques for mobile devices: A comparison between iOS and Android. International Journal of Pervasive Computing and Communications Vol. 11 No. 1, 2015. С. 102-126. URL: https://doi.org/10.1108/IJPCC-01-2015-0001 (дата звернення: 25.05.2025)
11. Apple Inc. Документація Timer. [Електронний ресурс] / Apple Inc. URL: https://developer.apple.com/documentation/foundation/timer (дата звернення: 25.05.2025)
12. Adegoke A. Performance Considerations When Using SwiftUI. URL: https://medium.com/@77adekunle/performance-considerations-when-using-swiftui-594202bb1d2e (дата звернення: 25.05.2025)
13. Hudson, P. How to add a shadow to a UIView [Електронний ресурс] / Paul Hudson // Hacking with Swift. URL: https://www.hackingwithswift.com/example-code/uikit/how-to-add-a-shadow-to-a-uiview (дата звернення: 25.05.2025)
14. Apple Inc. Creating Custom iOS User Interfaces [Електронний ресурс] / Apple Inc. URL: https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=http://devstreaming-cdn.apple.com/videos/wwdc/2014/221xxobzcm2j26x/221/221_creating_custom_ios_user_interfaces.pdf%3Fdl%3D1&ved=2ahUKEwj4nb6tjbyNAxU0CRAIHUJ_K-MQFnoECCQQAQ&usg=AOvVaw0FJF-u-jPv4hRuCPrUamqk (дата звернення: 25.05.2025)

15. Apple Inc. Advanced Graphics and Animations for iOS Apps. [Електронний ресурс] / Apple Inc. URL: https://docs.huihoo.com/apple/wwdc/2014/419_advanced_graphics_and_animation_performance.pdf (дата звернення: 25.05.2025)
16. Tran, A. Re-visiting reduce transparency in macOS [Електронний ресурс] / Andy Tran. URL: https://andytran93.com/2022/07/24/re-visiting-reduce-transparency-in-macos/ (дата звернення: 25.05.2025)
17. Rumi M. A., Asaduzzaman, Hasibul Hasan D. M., CPU power consumption reduction in android smartphone. 3rd International Conference on Green Energy and Technology (ICGET), Dhaka, Bangladesh, 2015. С. 1-6. URL: 10.1109/ICGET.2015.7315080 (дата звернення: 25.05.2025)
18. Palomba F., Di Nucci D., Panichella A., Zaidman A., De Lucia A. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology,Vol. 105, 2019. С. 43–55. URL: https://doi.org/10.1016/j.infsof.2018.08.004 (дата звернення: 25.05.2025)
19. Couto M., Saraiva J., Fernandes J. P. Energy Refactorings for Android in the Large and in the Wild. 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020. С. 217-228. URL: https://doi.org/10.1109/SANER48275.2020.9054858 (дата звернення: 25.05.2025)
20. Li D., Halfond W. G. J. An investigation into energy-saving programming practices for Android smartphone app development. GREENS 2014: Proceedings of the 3rd International Workshop on Green and Sustainable Software, 2014. С. 46–53. URL: https://doi.org/10.1145/2593743.2593750 (дата звернення: 25.05.2025)
21. Cruz L., Abreu R. Performance-Based Guidelines for Energy Efficient Mobile Applications. 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Buenos Aires, Argentina, 2017. С. 46-57. URL: https://doi.org/10.1109/MOBILESoft.2017.19 (дата звернення: 25.05.2025)

22. Cruz L., Abreu R. Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. CIbSE XXI Ibero-American Conference on Software Engineering, Colombia, 2018. URL: https://luiscruz.github.io/papers/cruz2018using.pdf (дата звернення: 25.05.2025)
23. Nguyen M.D., Huynh T.Q., Nguyen T.H. Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint. Huynh, VN., Inuiguchi, M., Le, B., Le, B., Denoeux, T. (eds) Integrated Uncertainty in Knowledge Modelling and Decision Making, 2016. URL: https://doi.org/10.1007/978-3-319-49046-5_29 (дата звернення: 25.05.2025)
24. Park J-J., Hong J.E., Lee S-H. Investigation for Software Power Consumption of Code Refactoring Techniques. Proceedings of the Twenty Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE), Vancouver, BC, Canada, 2014. C. 717–722, 2014. URL: https://scispace.com/pdf/investigation-for-software-power-consumption-of-code-20sa7ta5vt.pdf (дата звернення: 25.05.2025)
25. Jaing H., Yang H., Qin S., Su S. Detecting Energy Bugs in Android Apps Using Static Analysis. International Conference on Formal Engineering Methods, 2017. URL: http://dx.doi.org/10.1007/978-3-319-68690-5_12 (дата звернення: 25.05.2025)
26. nhrysiuk/EnergyUsageAnalyzer. GitHub. URL: https://github.com/nhrysiuk/EnergyUsageAnalyzer (дата звернення: 25.05.2025)