УДК 004.415:004.89

ОЦІНКА ЗДАТНОСТІ LLM ВИЯВЛЯТИ ТА УСУВАТИ БАЗОВІ «ЗАПАХИ КОДУ» У C#-КОДІ

Фуштор А. В. [1], Пенко В.Г. [2], Гунченко А.Ю. [3]
[1,2,3] Одеський національний університет імені І. І. Мечникова
[1,2,3] Україна, Одеса
[1]fushtor.anastasiia@stud.onu.edu.ua; [2]vpenko@onu.edu.ua;
[3]gunchenko.andrii@stud.onu.edu.ua

Анотація. *Оцінено ефективність LLM у задачах детекції та рефакторингу базових «запахів коду» в C#-проєктах на вибірці з 12 фрагментів. Модель виявила всі наявні дефекти (точність і повнота 100%), а також забезпечила 100% успішність компіляції та збереження функціональної поведінки за тестами. Отримані результати підтверджують ефективність великих мовних моделей, але вказують на необхідність контролю результатів розробником.*
Ключові слова: *великі мовні моделі; запахи коду; рефакторинг; мова програмування C#; цикломатична складність.*

Вступ. У сучасній інженерії програмного забезпечення етап супроводу є найбільш тривалим і витратним: на супровід та модифікацію вже створених систем припадає від 60% до 80% загальних витрат організацій-розробників [2]. Однією з головних причин цього є накопичення технічного боргу через «запахи коду» – ознаки слабкого дизайну програми, які не завжди є помилками, але ускладнюють супровід [1]. Їх усувають рефакторингом – зміною внутрішньої структури програми без зміни її зовнішньої поведінки [2], що полегшує читання та зміну модуля [1].

Класичні засоби статичного аналізу (Roslyn Analyzers, SonarQube) та метрики складності, зокрема цикломатична складність за Маккейбом [3], виявляють порушення заданих правил і порогів, а усунення «запахів», особливо пов'язаних із залежностями між класами, зазвичай виконується вручну. Великі мовні моделі (LLM, Large Language Models) на базі трансформерної архітектури здатні генерувати працездатний код, проте, як наголошують автори, він може лише здаватися правильним, тому потрібен людський контроль. Актуальність дослідження зумовлена необхідністю практичної оцінки здатності LLM не лише ідентифікувати базові «запахи коду», а й виконувати синтаксично та логічно коректний рефакторинг, правильність якого потребує перевірки тестами.

Отже, **мета роботи** – оцінити здатність великої мовної моделі виявляти та усувати базові «запахи коду» в C#-коді зі збереженням функціональної поведінки.

Основна частина. Для дослідження сформовано вибірку з 12 фрагментів коду мовою C# (по 3 на кожен із 4 типів «запахів») обсягом 10–50 рядків на основі типових задач обробки замовлень, нарахування знижок та відправки сповіщень. До вибірки увійшли такі дефекти [1,2]:

а) Long Method – методи, що перевищують обсяг одного екрана (понад 50–100 рядків) та виконують кілька незв'язаних функцій;

б) Duplicate Code – дублювання однакових блоків логіки (порушення принципу DRY, Don't Repeat Yourself);

в) Magic Numbers – явні числові константи у виразах та умовах замість іменованих констант чи переліків;

г) Tight Coupling – створення класом конкретних залежностей через new замість абстракцій (порушення принципу інверсії залежностей).

Для кожного фрагмента заздалегідь складено перелік наявних «запахів коду».

Досліджено модель – Claude Sonnet 5.5. Усі фрагменти передавалися з єдиним промптом, у якому тип «запаху» не названо: «Ти – досвідчений C#-розробник. Знайди в наведеному коді запахи коду та усунь їх рефакторингом. Спочатку назви кожен знайдений запах, потім поверни повний виправлений код. Збережи зовнішню поведінку коду незмінною, включно з текстами повідомлень.»

Оцінювання виконувалося за чотирма критеріями:

а) точність і повнота виявлення: Precision – частка правильно названих «запахів» серед усіх названих, Recall – частка знайдених «запахів» серед наявних:

$$P = \frac{TP}{(TP+FP)},\ R = \frac{TP}{(TP+FN)}, \quad (1)$$

де TP – правильно виявлені «запахи», FP – хибні спрацьовування, FN – пропущені «запахи»[4];

б) успішність компіляції – частка рефакторингів, які компілюються без помилок;

в) збереження поведінки – частка рефакторингів, для яких проходять усі заздалегідь написані тести (для Tight Coupling тести адаптовано до інтерфейсів, створених моделлю);

г) зміна цикломатичної складності на рівні класу, обчисленої інструментом Visual Studio Code Metrics за формулою Маккейба $V(G) = e - n + 2p$ [3], де e – кількість ребер графа потоку управління, n – кількість вузлів, p – кількість компонент зв'язності; фактично V(G) дорівнює кількості точок прийняття рішень плюс один [1]. Відносну зміну складності визначають як

$$\Delta V(G) = \frac{V_{\text{після}} - V_{\text{до}}}{V_{\text{до}}} \times 100\%, \quad (2)$$

де $V_{\text{до}}$ – початкова цикломатична складність коду до рефакторингу, $V_{\text{після}}$ –цикломатична складність коду після рефакторингу за допомогою LLM.

Як приклад розглянуто фрагмент із групи Tight Coupling (лістинг 1). Клас OrderProcessor розраховує знижку, записує лог і надсилає SMS; залежності FileLogger і SmsNotificationService створюються безпосередньо через new, тому їх не можна підмінити в модульному тесті.

```
public class OrderProcessor
{
    public void ProcessOrder(decimal totalAmount, string customerType)
    {
        // ... обчислення знижки та finalPrice.
        FileLogger logger = new FileLogger();
        logger.Log("Processing payment for amount: " + finalPrice);
        SmsNotificationService sms = new SmsNotificationService();
        sms.SendSms("Your order total is " + finalPrice);
    }
}
```

Лістинг 1 – Фрагмент коду з Tight Coupling до рефакторингу

Після усунення Tight Coupling (лістинг 2) залежності передаються через конструктор за інтерфейсами, а тексти повідомлень збережено без змін.

```
public interface ILoggerService { void Log(string message); }
public interface INotificationService
{ void Notify(string message); }
public class OrderProcessor
{
    private readonly ILoggerService _logger;
    private readonly INotificationService _notifier;
    public OrderProcessor(ILoggerService logger,
                          INotificationService notifier)
    {
        _logger = logger;
        _notifier = notifier;
    }
    public void ProcessOrder(decimal totalAmount, string customerType)
    {
        // ... обчислення знижки та finalPrice
        _logger.Log("Processing payment for amount: " + finalPrice);
        _notifier.Notify("Your order total is " + finalPrice);
    }
}
```

Лістинг 2 – Фрагмент коду з Tight Coupling після рефакторингу

Складність методу ProcessOrder не змінилась (V(G) = 5 до і після), бо усунення Tight Coupling змінює залежності, а не розгалуження; обсяг коду зріс на два інтерфейси та конструктор. У групі Magic Numbers модель замінила літерали іменованими константами та додатково виділила окремий метод, що виходить за межі усунення цього «запаху». Підсумки за типами «запахів» наведено в таблиці 1. За точністю, повнотою, успішністю компіляції та збереженням поведінки результати однакові для всіх типів (100%).

Таблиця 1 – Результати виявлення та усунення «запахів коду»

Тип «запаху»	ΔV(G), %	Пояснення
Long Method	+66,7	Виділення методів збільшує суму V(G) класів, хоча максимальна складність одного методу зменшилась (−44,4%)
Duplicate Code	−20,0	Спільні методи зменшили складність
Magic Numbers	0,0	Винесення констант не змінює кількість гілок
Tight Coupling	+60,0	Складність росте лише через конструктори; тести потрібно адаптувати до інтерфейсів

Як видно з таблиці 1, на вибірці з 12 фрагментів модель виявила основний «запах» у всіх фрагментах, а також супутні Magic Numbers у двох фрагментах з інших груп (точність і

повнота 100%). Усі рефакторинги компілюються й проходять тести. Складність класу зростає там, де рефакторинг додає допоміжні методи чи конструктори (Long Method, Tight Coupling), знижується при усуненні дублювання і не змінюється при заміні літералів константами.

Висновки. На вибірці з 12 фрагментів Claude Sonnet 5.5 виявила всі наявні «запахи коду» без хибних спрацьовувань (точність і повнота 100%), у тому числі супутні Magic Numbers у двох фрагментах з інших груп. Усі згенеровані рефакторинги скомпілювались і зберегли поведінку (100%), проте для Tight Coupling тести довелося адаптувати до інтерфейсів, створених моделлю. Зміна V(G) на рівні класу залежала від типу «запаху»: +66,7% для Long Method, −20,0% для Duplicate Code, 0,0% для Magic Numbers і +60,0% для Tight Coupling. Зростання пов'язане з появою допоміжних методів і конструкторів, тому ця метрика сама по собі не відображає якість рефакторингу. Результати показують, що LLM придатна як допоміжний засіб рефакторингу, але потребує перевірки тестами й оглядом коду розробником.

Використання інструментів ШІ. При підготовці матеріалу використовувалася велика мовна модель Claude Sonnet 5.5 з метою структурування результатів експерименту у вигляді порівняльної таблиці, а також генерації англомовного блоку тез у розділі «Abstract».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd ed.). Addison-Wesley Professional.
[2] Пенко, В. Г. (2026). *Інженерія програмного забезпечення : конспект лекцій* [Електронний ресурс]. Одеса : Одеський національний університет імені І. І. Мечникова, 150 с.
[3] McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308–320.

Fushtor A. V., Penko V. H., Hunchenko A. Yu.

Evaluating the ability of LLMs to detect and remove basic code smells in C# code

Abstract: The effectiveness of LLMs in detecting and refactoring basic code smells in C# projects was evaluated on a sample of 12 code snippets. The model detected all present smells (100% precision and recall) and achieved 100% compilation success and functional behavior preservation verified by unit tests. The obtained results confirm the efficiency of large language models, while highlighting the need for developer supervision over the generated output.

Keywords: large language models; code smells; refactoring; C#; cyclomatic complexity.