



ОЦІНЮВАННЯ ВПЛИВУ СКЛАДНОСТІ РОЗГАЛУЖЕНЬ C#-КОДУ НА ТРИВАЛІСТЬ ПІДГОТОВКИ ТЕСТІВ ВРУЧНУ ТА ЗАСОБАМИ ШТУЧНОГО ІНТЕЛЕКТУ

Виявлено практичну потребу в кількісному оцінюванні того, як складність розгалужень програмного коду впливає на трудомісткість підготовки тестів у C#-проектах. Встановлено, що для такого оцінювання недостатньо враховувати тільки розмір програмного файлу або факт наявності складних тестів, оскільки реальні витрати часу та ресурсів на тестування C#-коду залежать від кількості альтернативних шляхів його виконання, глибини вкладеності умовних конструкцій, кількості методів тестування, обсягу тестового коду та кількості його перевірок. З'ясовано, що складність розгалужень C#-коду безпосередньо пов'язана з кількістю сценаріїв його виконання, які потрібно продумати під час модульного тестування коду: кожна додаткова умова, цикл або вкладений блок збільшує кількість ситуацій, для яких потрібно підібрати вхідні дані, визначити очікуваний результат і сформулювати перевірку. Оцінено тривалість підготовки тестів за двома підходами: ручним написанням і створенням тестів із використанням інструментів штучного інтелекту (ШІ). Охарактеризовано модель підготовки тестів вручну, згідно з яким тривалість їх розроблення залежить від кількості методів тестування, рядків тестового коду та вартості підготовки одного тесту. Охарактеризовано модель використання засобів ШІ, що додатково враховує тривалість формулювання запиту, уточнення контексту, генерування тестового коду, ревію, пошук граничних випадків і виправлення некоректних його перевірок. Встановлено, що перехід від низького до дуже високого рівня розгалуженості програмного коду супроводжується збільшенням орієнтовної тривалості ручної підготовки тестів приблизно від 20 до 159 хв, тобто майже у вісім разів. Виявлено, що застосування засобів ШІ скорочує тривалість підготовки тестів у всіх групах складності програмного коду, однак цей ефект зменшується зі зростанням складності коду: для простого коду економія часу становить близько 60 %, а для дуже складного – близько 35 %. З'ясовано, що причина такого зменшення часу полягає в перенесенні частини роботи з підготовки тестів на перевірку логіки C#-коду, коректності його тестових перевірок, відповідності бізнес-правилам і повноти сценаріїв його виконання. Практична цінність проведеного дослідження полягає в тому, що показники складності розгалужень C#-коду можна використовувати для планування процесу його тестування, оцінювання QA-завдань, пріоритетизації рефакторингу програмного коду та визначення його ділянок, де ШІ є корисним допоміжним інструментом, але не може замінити експертне оцінювання розробника.

Ключові слова: складність розгалужень C#-коду; тестові сценарії для C#-коду; тестові перевірки; рефакторинг програмного коду; трудомісткість підготовки тестів; модульне тестування C#-коду.

Вступ / Introduction

Забезпечення якості програмного коду є одним із центральних завдань інженерії програмного забезпечення (ПЗ), оскільки саме стан коду визначає здатність системи розвиватися без надмірного зростання витрат на її супровід. У великих і середніх проектах код постійно змінюється, доповнюється новими вимогами, проходить через code review, автоматизовані перевірки й тестування. За таких умов якість програмного коду неможливо оцінювати тільки за фактом успішної компіляції проекту або проходження відповідних тестів, адже складна структура розгалужень C#-коду може створювати приховані ризики для майбутніх змін.

У науковій літературі й практиці розроблення ПЗ для оцінювання складності коду використовують різні метрики: кількість рядків коду, цикломатичну його скла-

дність, кількість умовних операторів і циклів, показники вкладеності керувальних конструкцій та тестове покриття коду. Ці показники корисні, однак для прикладних завдань важливо не тільки зафіксувати складність програмного коду, а й визначити її вплив на процес тестування C#-коду. Якщо метод або програмний алгоритм має багато альтернативних шляхів його виконання, розробнику потрібно створити більше тестових сценаріїв виконання і перевірити більше граничних випадків.

Особливою проблемою є branching complexity, тобто складність розгалужень керівного потоку. Два фрагменти коду можуть мати однаковий розмір, але істотно відрізнятися за кількістю логічних умов, циклів, switch/case-конструкцій і вкладених блоків. Для тестування C#-коду така різниця принципова: кожна додаткова гілка коду потенційно збільшує кількість сценаріїв їх виконання, які потрібно продумати, реалізувати та перевірити.

© Copyright 2026 by the author(s)

Сенів Максим Михайлович, канд. техн. наук, доцент, кафедра програмного забезпечення.

Email: maksym.m.seniv@lpnu.ua; <https://orcid.org/0000-0003-1044-4628>

Іншин Валерій Валерійович, студент, кафедр програмного забезпечення.

Email: valerii.inshyn.mpzip.2025@lpnu.ua; <https://orcid.org/0009-0007-3166-7923>

Цитування за ДСТУ: Сенів М. М., Іншин В. В. Оцінювання впливу складності розгалужень C#-коду на тривалість підготовки тестів вручну та засобами штучного інтелекту. Науковий вісник НЛТУ України. 2026, т. 36, № 4. С. 36–43.

Citation APA: Seniv, M. M., & Inshyn, V. V. (2026). Assessing the impact of C# branching complexity on test writing time manually and with ai assistance. *Scientific Bulletin of UNFU*, 36(4), 36–43. <https://doi.org/10.36930/40360404>

Актуальність дослідження полягає в необхідності встановити, як складність розгалужень виробничого коду пов'язана з тривалістю підготовки тестів. Особливої уваги потребує порівняння ручного підходу до підготовки тестів та тестування з використанням інструментів штучного інтелекту, оскільки сучасні інструменти ШІ здатні швидко генерувати тестовий код, але не усувають потреби людського ревію, виправлення помилок і перевірки повноти сценаріїв виконання.

Об'єкт дослідження – оцінювання впливу складності розгалужень C#-коду на тривалість підготовки тестів.

Предмет дослідження – методи і засоби оцінювання впливу складності розгалужень C#-коду на тривалість підготовки тестів вручну та програмно, що дасть змогу порівняти трудомісткість традиційного та автоматизованого розроблення тестового коду.

Мета роботи – розробити моделі тривалості підготовки тестів вручну і засобами ШІ, які дадуть змогу обґрунтувати доцільність застосування показників branching complexity під час оцінювання QA-завдань, пріоритетизації рефакторингу програмного коду та вибору його ділянок для AI-assisted testing.

Для досягнення зазначеної мети визначено такі основні завдання дослідження:

- охарактеризувати теоретичні підходи до оцінювання складності розгалужень C#-коду, що дасть змогу визначити основні метрики, придатні для кількісного аналізу структури керівного потоку програмного коду;
- сформувати набір пар виробничих і тестових файлів і порівняти тестові показники для різних рівнів branching complexity, що дасть змогу встановити зв'язок між складністю розгалужень C#-коду і фактичним обсягом його тестування;
- побудувати модель тривалості ручної підготовки тестів і модель тривалості AI-assisted testing, що дасть змогу порівняти трудомісткість традиційного та автоматизованого підтриманого підходів до розроблення тестового коду;
- визначити практичне значення отриманих результатів для планування процесу тестування програмного коду та використання засобів ШІ для розроблення тестів, що дасть змогу обґрунтувати доцільність застосування показників branching complexity під час оцінювання QA-завдань, пріоритетизації рефакторингу програмного коду та вибору його ділянок для AI-assisted testing.

Аналіз останніх досліджень та публікацій. Оцінювання впливу складності розгалужень (цикломатичної складності) у C#-кодi на тривалість підготовки тестів має принципові відмінності для ручного тестування та генерування тестів засобами ШІ (наприклад, GitHub Copilot, ChatGPT, спеціалізовані AI-плагіни).

У роботі [7] подано модель якості програмного забезпечення ISO/IEC 25010:2023, у межах якої підтримуваність, надійність і тестованість визначено як важливі характеристики програмного продукту. У цьому стандарті якість ПЗ розглянуто не тільки як відповідність функціональним вимогам, а і як здатність системи до подальшого супроводу, перевірки та розвитку. Це положення є важливим для цього дослідження, оскільки складність розгалужень виробничого коду безпосередньо впливає на його тестованість, тобто на те, наскільки легко сформувати повний і коректний набір тестових сценаріїв виконання.

У роботі [10] запропоновано метрику цикломатичної складності програмного коду, яка ґрунтується на аналізі графа керування потоком програми. Автор показав, що кількість незалежних шляхів її виконання може

бути використано для кількісного оцінювання складності програмного коду. Для цього дослідження зазначена метрика є базовою, оскільки вона дає змогу пов'язати кількість умов, циклів і розгалужень C#-коду з потенційною кількістю тестових сценаріїв виконання. Водночас, цикломатична складність програмного коду не повністю відображає трудомісткість процесу тестування C#-коду, оскільки не враховує всіх практичних витрат на підготовку тестових даних, ревію та перевірку граничних випадків.

У роботі [12] розглянуто основні принципи тестування ПЗ та наголошено на потребі перевірки не тільки основного сценарію його виконання, а й альтернативних шляхів виконання, помилкових вхідних даних, граничних значень і виняткових ситуацій. Такий підхід підтверджує, що кожна додаткова гілка у програмному кодi може створювати окремий тестовий випадок. Отже, складність розгалужень коду впливає не тільки на загальний обсяг тестового набору сценаріїв його виконання, а й на кількість перевірок умов виконання, які потрібно реалізувати для підтвердження очікуваної поведінки програми.

У роботі [18] проаналізовано, чи є покриття гілок достатньо надійним показником ефективності тестування C#-коду. Автори показали, що branch coverage краще відображає перевірку логіки роботи програми, ніж звичайне покриття рядків коду, однак саме собою не гарантує повної якості тестового набору сценаріїв виконання. Це важливо для нашого дослідження, оскільки складність розгалужень коду потрібно розглядати не ізольовано, а у зв'язку з кількістю методів його тестування, обсягом тестового коду та кількістю assertions.

У роботі [3] узагальнено практики модульного тестування C#-коду та основні проблеми, з якими стикаються розробники під час підготовки тестів. Автори зазначили, що труднощі виникають не тільки під час написання тестового коду, а й під час вибору тестових сценаріїв виконання, підготовки вхідних даних і підтримки тестового набору сценаріїв виконання. Це підтверджує доцільність використання таких показників, як кількість методів тестування, Test LOC та assertions, для оцінювання фактичної трудомісткості процесу тестування програмного коду з різним рівнем його розгалуженості.

У роботі [5] представлено інструмент EvoSuite, призначений для автоматичного генерування наборів тестів для об'єктно-орієнтованого ПЗ. Автори показали, що автоматизація може зменшувати обсяг рутинної роботи під час підготовки тестів, однак якість отриманих тестів потребує додаткового оцінювання. Це положення є важливим для цього дослідження, оскільки AI-assisted testing також не можна розглядати як повністю автономну заміну розробника: згенерований тестовий код потребує перевірки, виправлення та узгодження з реальною логікою системи.

У роботі [13] досліджено вплив інтелектуального помічника для програмування GitHub Copilot на продуктивність розробників. Автори встановили, що інструменти ШІ можуть пришвидшувати виконання завдань програміста, однак очікуваний ефект залежить від складності завдання, якості контексту та здатності розробника оцінити правильність результату. Це узгоджується з тематикою нашого дослідження, оскільки під час підготовки тестів за допомогою ШІ важливо враховувати не

тільки тривалість генерування коду, а й час формулювання запиту, ревію, пошук пропущених сценаріїв виконання і виправлення некоректних assertions.

У роботі [2] проаналізовано автоматизоване генерування тестів із використанням великих мовних моделей. Автори показали, що такі моделі здатні швидко створювати тестові методи, однак їхні результати залежать від якості вхідного контексту, складності коду та повноти опису очікуваної його поведінки. Для цього дослідження зазначена робота є важливою, оскільки підтверджує потребу враховувати додатковий час на перевірку AI-generated тестів, особливо у разі складного коду з великою кількістю розгалужень коду.

У роботі [1] систематизовано основні підходи до тестування ПЗ та розглянуто методи побудови тестових сценаріїв виконання для перевірки різних шляхів виконання програми. Автори наголошують, що якісне тестування програмного коду має враховувати не тільки типові вхідні дані, а й граничні значення, альтернативні сценарії та помилкові ситуації. Це положення є важливим для цього дослідження, оскільки складність розгалужень C#-коду безпосередньо збільшує кількість сценаріїв їх виконання, які потрібно продумати, реалізувати й перевірити під час розроблення модульних тестів.

У роботі [4] досліджено вплив GitHub Copilot як інструменту AI-assisted programming на процес розроблення ПЗ. Автори проаналізували переваги та ризики використання засобів ШІ-помічника і показали, що такі інструменти можуть пришвидшувати написання коду, однак їхній результат потребує перевірки з боку розробника. Для нашого дослідження зазначене джерело є важливим, оскільки підтверджує, що під час підготовки тестів із використанням ШІ потрібно враховувати не тільки тривалість генерування коду, а й час на ревію, виправлення помилок, перевірку assertions і повноту тестових сценаріїв виконання.

У роботі [9] розглянуто принципи написання зрозумілого, підтримуваного та структурованого програмного коду. Автор вказує на те, що надмірно складний код, великі методи, глибока вкладеність і нечітка його структура ускладнюють розуміння, супровід і подальше розширення програмної системи. Це безпосередньо пов'язано з темою цього дослідження, оскільки C#-код із високою складністю розгалужень є важчим не тільки для читання й модифікації, а й для його тестування, адже потребує більшої кількості методів тестування, детальніших перевірок результатів виконання коду і додаткового часу на аналіз поведінки програми.

Аналіз останніх досліджень і публікацій свідчить, що в науковій літературі достатньо висвітлено питання якості ПЗ, його цикломатичної складності програмного коду, branch coverage, модульного тестування та авто-

матизованого генерування тестів. Проте, практичне питання кількісного порівняння тривалості ручного та AI-assisted підготовки тестів залежно від рівня складності розгалужень C#-коду залишається недостатньо дослідженим як на сьогодні. Саме тому проведення цього дослідження є доцільним, оскільки воно поєднує аналіз структурної складності production-коду з оцінюванням відповідних тестових показників і розрахункової трудомісткості підготовки тестів.

Матеріали та методи дослідження. Дослідження здійснено на основі відкритих C#-проектів, для яких сформовано вибірку зіставлених пар виробничих і тестових файлів. Для визначення впливу складності розгалужень C#-коду на трудомісткість його тестування проаналізовано структурні показники production-коду та відповідні характеристики наборів тестових сценаріїв його виконання. Під час дослідження використано методи статичного аналізу програмного коду, кількісного порівняння тестових метрик, групування програмних файлів за рівнем branching complexity, а також розрахункового моделювання тривалості ручної підготовки тестів і підготовки тестів із використанням інструментів ШІ. Статистичне опрацювання отриманих результатів виконано у спосіб порівняльного аналізу значень між групами складності програмного коду.

Результати дослідження та їх обговорення / Research results and their discussion

Формування вибірки та одиниця аналізу. Одиницею аналізу дослідження є зіставлена пара "виробничий файл – тестовий файл". Для кожного виробничого файлу Foo.cs здійснювався пошук відповідного тестового файлу FooTests.cs або FooTest.cs. Такий підхід до формування вибірки не охоплює всіх можливих стилів організації тестів, але забезпечує відтворювану процедуру і дає змогу пов'язати складність production-коду з фактичним обсягом тестів.

До фінального масиву для оцінювання трудомісткості тестування увійшли 229 зіставлених пар. Для виробничого коду фіксувалися показники його розгалуженості та розміру, зокрема – цикломатична складність, кількість рядків і максимальна вкладеність керівних конструкцій. Для тестового коду фіксувалися кількість методів тестування, кількість рядків тестового коду та кількість assertions. Саме ці тестові показники часто використовують для переходу від структурної складності коду до оцінювання тривалості його тестування (табл. 1). Головним показником є цикломатична складність (англ. *Cyclomatic Complexity*) тестового коду, яка безпосередньо вимірює кількість лінійно незалежних маршрутів через програмний код.

Табл. 1. Показники, які використано для оцінювання впливу складності розгалужень C#-коду на його тестування / Metrics used to assess the impact of the complexity of C# code branching on its testing

Категорія показників	Показники	Призначення в аналізі
Branching у виробничому коді	CC, branch count, max nesting depth	Характеризують кількість і структуру альтернативних шляхів виконання.
Обсяг тестового набору сценаріїв виконання	TestMethods, TestLOC, Assertions	Показують, скільки тестових сценаріїв виконання і перевірок потрібно для коду.
Тривалість ручного написання тесту	T_manual	Оцінює трудомісткість підготовки тестів без генеративних інструментів.
Тривалість написання тесту із використанням ШІ	T_AI	Оцінює тривалість генерування, перевірки та виправлення AI-generated тестів.
Відносна економія часу	Saving, %	Показує, наскільки AI-assisted підхід скорочує тривалість підготовки тестів порівняно з їх ручним написанням.

Групування коду за рівнем branching complexity.

Для порівняння результатів аналізу файли поділено на чотири групи за зростанням branching complexity. У тексті вони позначені як низький, помірний, високий і дуже високий рівень розгалуженості програмного коду. Такий поділ дає змогу порівняти поведінку тестових показників без прив'язки до однієї конкретної метрики. Важливим є не саме числове значення окремого показника, а тенденція: зі зростанням кількості й складності гілок зростає кількість сценаріїв їх виконання, які має перевірити тестовий набір (табл. 2).

Табл. 2. Обсяг тестування C#-коду за групами branching complexity / The extent of C# code testing by branching complexity groups

Рівень branching complexity	TestMethods	TestLOC	3 x TestMethods	TestLOC / 5	T_manual, хв
Низький	57	4,0	41,6	4,8	10,4
Помірний	57	6,7	73,2	8,5	10,9
Високий	57	13,4	174,7	33,3	13,0
Дуже високий	58	23,5	452,2	71,3	19,2

Примітка: N – кількість проаналізованих зіставлених пар "виробничий файл – тестовий файл" у відповідній групі.

Результати аналізу показують, що зі зростанням branching complexity збільшується і кількість методів тестування, і обсяг тестового коду, і кількість перевірок тестових сценаріїв його виконання. У групі very high branching середня кількість методів тестування становить 23,5 проти 4,0 у групі low branching. Обсяг тестового коду зростає від 41,6 до 452,2 рядка, а кількість assertions – від 4,8 до 71,3. Отже, складніший branching потребує не тільки більшої кількості тестів, а й більш деталізованих перевірок очікуваних результатів виконання коду в межах цих тестів (рис. 1).

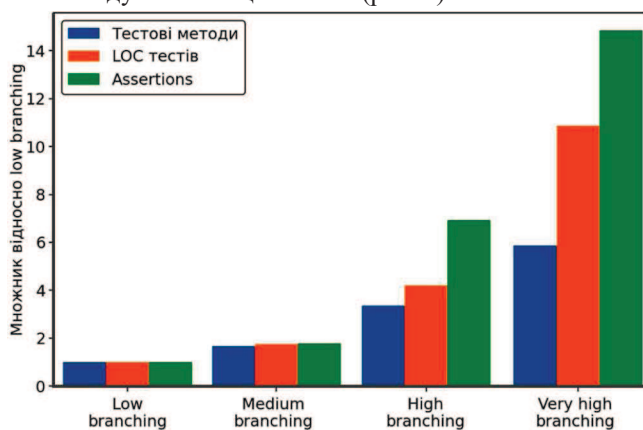


Рис. 1. Зростання обсягу тестування C#-коду зі збільшенням branching complexity / The increase in the volume of C# code testing as branching complexity increases

Модель тривалості ручної підготовки тестів.

Для оцінювання тривалості ручної підготовки тестів використано модель, що складається з двох компонентів: базового часу на розроблення кожного тестового методу та часу на написання рядків тестового коду. Базовий час охоплює формування ідеї тесту, підготовку вхідних даних, назву методу, виклик перевірюваної логіки та мінімальну перевірку результату, а саме

$$T_{\text{manual}} = tct \cdot \text{TestMethods} + \text{TestLOC} / cl, \quad (1)$$

де: T_{manual} – орієнтовна тривалість ручної підготовки тестів, хв; TestMethods – кількість методів тестування;

TestLOC – кількість рядків тестового коду. Значення tct (test creation time) беремо як змінну, що оцінює мінімальну організаційну вартість розроблення одного тесту, хв, а cl (code lines) – як умовну швидкість ручного написання тестового коду в рядках за хвилину. Модель не претендує на точне вимірювання індивідуальної продуктивності конкретного програміста, але забезпечує однакове правило порівняння між групами. Для експерименту приймаємо $tct = 3$ хв, тобто мінімальна тривалість аналізу методу, продумування сценарію, підготовки тестових даних і написання тесту (табл. 3).

Також візьмемо $cl = 5$, як мінімальне значення кількості рядків коду для написання тестового методу: ініціалізація методу – 1 рядок, підготовка даних – 1 рядок, виконання сценарію – 2 рядки, перевірка – 1 рядок (див. табл. 3). За цією моделлю тривалість ручної підготовки тестів зростає приблизно від 20 хв – для низького branching до 161 хв – для дуже високого branching. Отже, високий branching збільшує трудомісткість процесу тестування C#-коду майже у 8 разів. Це зростання можна пояснити двома причинами: по-перше, потрібно створити більше окремих тестових сценаріїв виконання; по-друге, кожен тест стає довшим, оскільки потребує підготовки більшої кількості даних і, як наслідок, більшої кількості його перевірок.

Табл. 3. Результати оцінювання тривалості ручної підготовки тестів / Results of the assessment of the duration of manual test preparation

Рівень branching complexity	TestMethods	TestLOC	3 x TestMethods	TestLOC / 5	T_manual, хв
Низький	4,0	41,6	12,0	8,3	20,3
Помірний	6,7	73,2	20,1	14,6	34,7
Високий	13,4	174,7	40,2	34,9	75,1
Дуже високий	23,5	452,2	70,5	90,4	160,9

Модель тривалості підготовки тестів із використанням ШІ. Для AI-assisted підходу підготовки тестів із використанням ШІ тривалість цього процесу не можна зводити тільки до швидкості генерування коду. Навіть якщо інструмент ШІ швидко створює тестовий метод, розробник має перевірити коректність сценарію, відповідність бізнес-логіці, повноту edge cases, стабільність тесту та відсутність помилкових assertions. Тому модель тривалості підготовки тестів з використанням ШІ складається з трьох частин: базовий час взаємодії з інструментом, тривалість генерування тестового коду та тривалість перевірки й виправлення, а саме:

$$T_{AI} = T_{\text{int}} + T_{\text{gen}} + T_{\text{refix}}, \quad (2)$$

$$T_{AI} = h \cdot \text{TestMethods} + \frac{\text{TestLOC}}{20} + k_{\text{branch}} + \text{TestMethods} = \text{TestMethods} \cdot (h + 1) + \text{TestLOC} \cdot 0,05 + k_{\text{branch}}, \quad (3)$$

де h – середня тривалість взаємодії із засобом ШІ для одного тестового методу, що охоплює формулювання запиту, уточнення контексту та первинне рев'ю тесту, хв/тест. У наших розрахунках прийнято $h = 1$ хв/тест. У формулі (3) компонент $h \cdot \text{TestMethods}$ враховує тривалість запиту, уточнення контексту та первинне рев'ю тесту, а $\text{TestLOC} / 20$ – швидше отримання тестового коду порівняно з ручним написанням. Коефіцієнт k_{branch} відображає додатковий час на перевірку й виправлення згенерованого тестового коду залежно від складності branching: 0,5; 0,8; 1,4 і 2,5 хв на тест для груп low, medium, high і very high (табл. 4).

Табл. 4. Результати оцінювання тривалості підготовки тестів із використанням ШІ / Results of an assessment of the time taken to create tests using AI

Рівень branching complexity	Взаємодія, хв	Генеравання, хв	Рев'ю і виправлення, хв	T_AI, хв	Економія часу, %
Низький	4,0	2,1	2,0	8,1	60
Помірний	6,7	3,7	5,4	15,7	55
Високий	13,4	8,7	18,8	40,9	46
Дуже високий	23,5	22,6	58,8	104,9	35

Розрахунок показує, що AI-assisted підхід до підготовки тестів із використанням ШІ скорочує тривалість їх підготовки у всіх групах, але ефект не є однаковим (рис. 2). Для low branching очікувана економія часу становить близько 60 %, тоді як для very high branching – близько 35 % (рис. 3). Це означає, що засоби ШІ найкраще пришвидшують рутинне розроблення простих і помірних тестів, але його перевага зменшується там, де код має багато гілок і потребує уважного аналізу сценаріїв виконання.

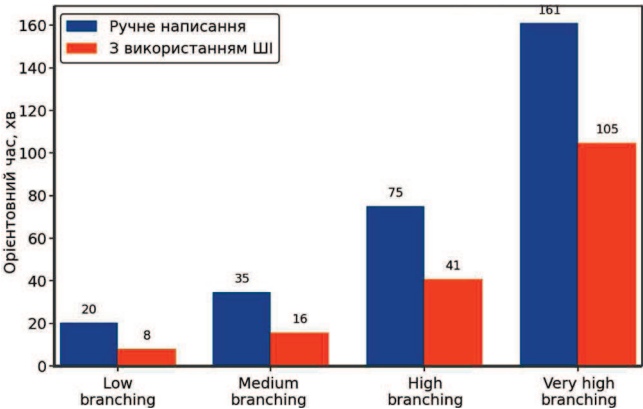


Рис. 2. Порівняння тривалості підготовки тестів вручну та з використанням засобів ШІ / A comparison of the time taken to prepare tests manually and using AI

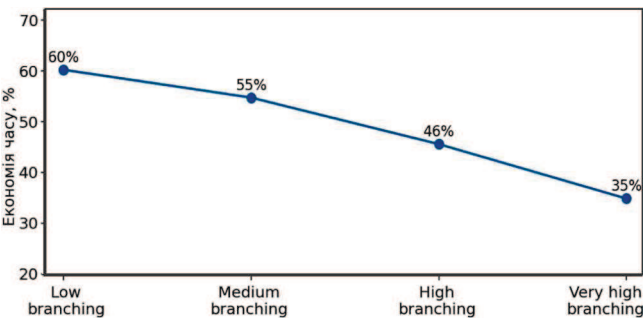


Рис. 3. Очікувана економія часу за використання засобів ШІ залежно від складності branching / Expected time savings when using AI tools, depending on the complexity of the branching

Результати дослідження підтверджують, що зі зростанням складності розгалужень у виробничому коді істотно збільшується обсяг тестування C#-коду (табл. 5). У групі з низьким branching середня кількість методів тестування становить 4,0, тоді як у групі з дуже високим branching – 23,5. Отже, кількість методів тестування зростає приблизно у 5,9 раза. Це означає, що складніший код потребує не просто більшої кількості його перевірок, а більшої кількості окремих тестових сценаріїв виконання.

Подібна тенденція спостерігається і для обсягу тестового коду. Для низького branching середній Test LOC становить 41,6 рядка, а для дуже високого – 452,2 ряд-

ка. Тобто, обсяг тестового коду збільшується приблизно у 10,9 раза. Таке зростання можна пояснити тим, що тести для складного коду потребують більшої підготовки вхідних даних, додаткового налаштування об'єктів, перевірки альтернативних результатів і більш детальних assertions.

Кількість assertions зростає ще інтенсивніше: від 4,8 – у групі низького branching до 71,3 – у групі дуже високого branching. Це приблизно у 14,9 раза більше. Така динаміка свідчить, що складні розгалуження збільшують не тільки кількість методів тестування, а й кількість очікувань, які потрібно явно перевірити у тестах. Іншими словами, branching complexity збільшує як ширину тестового набору сценаріїв виконання, так і глибину їх перевірок усередині окремих тестів.

Табл. 5. Узагальнені результати впливу branching complexity на обсяг тестування C#-коду / Summary of the findings on the impact of branching complexity on the testing effort for C# code

Рівень branching complexity	TestMet hods	Test LOC	Assertions	Зростання Test LOC
Низький	4,0	41,6	4,8	1,0 x
Помірний	6,7	73,2	8,5	1,8 x
Високий	13,4	174,7	33,3	4,2 x
Дуже високий	23,5	452,2	71,3	10,9 x

Результати оцінювання тривалості ручної підготовки тестів показали, що їх трудомісткість зростає майже експоненційно відносно практичного навантаження на розробника. Для низького branching орієнтовна тривалість становить 20,3 хв, для помірного – 34,7 хв, для високого – 75,1 хв, а для дуже високого – 160,9 хв (табл. 6). Отже, перехід від простого до найскладнішого коду збільшує тривалість ручної підготовки тестів приблизно у 7,9 раза.

Порівняння з AI-assisted підходом показало, що використання засобів ШІ скорочує тривалість підготовки тестів у всіх групах, але величина економії часу залежить від складності коду. Для низького branching оцінена тривалість зменшується від 20,3 до 8,1 хв, тобто економія часу становить близько 60 %. Для помірного branching тривалість зменшується з 34,7 до 15,7 хв, тобто приблизно на 55 %. Для високого branching економія часу становить близько 46 %, а для дуже високого branching – близько 35 % (див. табл. 6).

Табл. 6. Порівняння тривалості ручного та AI-assisted підготовки тестів / A comparison of the duration of manual and AI-assisted test preparation

Рівень branching complexity	T_manual, хв	T_AI, хв	Економія часу, %	Інтерпретація
Низький	20,3	8,1	60	ШІ ефективно пришвидшує типові тести
Помірний	34,7	15,7	55	Потрібне помірне рев'ю
Високий	75,1	40,9	46	Зростає тривалість перевірки сценаріїв виконання
Дуже високий	160,9	104,9	35	Рев'ю і виправлення стають критичними

Отримані результати демонструють важливу закономірність: засоби ШІ добре скорочують рутинну частину підготовки тестів, але не повністю компенсують складність аналізу багатьох гілок виконання коду. Коли код має низьку або помірну кількість розгалужень C#-коду, згенеровані тести зазвичай легше перевірити й адаптувати. Коли ж код має дуже високий branching,

розробник витрачає більше часу на рев'ю, уточнення сценаріїв виконання, пошук пропущених edge cases і виправлення некоректних assertions.

Найважливішим результатом є те, що AI-assisted testing не усуває залежність між branching complexity та трудомісткістю процесу тестування C#-коду. Навіть за використання засобів ШІ тривалість підготовки тестів зростає від 8,1 хв – для низького branching до 104,9 хв – для дуже високого branching. Тобто, дуже складний код залишається складним для тестування C#-коду незалежно від способу розроблення тестового коду. Різниця полягає в тому, що ШІ переносить частину роботи з ручного написання коду тестових сценаріїв на перевірку, уточнення й виправлення.

Отже, результати дослідження дають можливість зробити практичний висновок: використання засобів ШІ є найбільш доцільним для пришвидшення типових і повторюваних тестів, але для коду з високою кількістю розгалужень потрібна посилена участь розробника. У таких випадках засоби ШІ доцільно розглядати як інструмент підготовки початкового варіанта тестів, а не як остаточне джерело перевіреного тестового набору сценаріїв виконання.

Отримані результати узгоджуються з теоретичним уявленням про те, що кожне розгалуження у коді створює додатковий шлях його виконання, а отже – потенційний тестовий сценарій. На практиці розробник рідко тестує всі можливі комбінації шляхів виконання, оскільки їхня кількість може швидко зростати. Проте, навіть вибір репрезентативних сценаріїв виконання вимагає часу: потрібно зрозуміти логіку умов, визначити важливі граничні випадки, підготувати вхідні дані та сформулювати очікувані результати.

Зростання TestMethods і Test LOC у групах із вищим branching показує, що тестовий код реагує на структурну складність production-коду. Це підтверджує доцільність використання branching complexity як одного з індикаторів планування процесу тестування програмного коду. Якщо програмний файл або модуль має велику кількість умов, циклів і альтернативних шляхів виконання, команда може заздалегідь очікувати, що його тестування потребуватиме більше часу, ніж тестування Простого лінійного коду аналогічного розміру.

Водночас, порівняння ручного та AI-assisted підходів до підготовки тестів із використанням ШІ показує, що ефективність засобів ШІ не є сталою. Простий код дає найбільшу частку економії часу, оскільки типові тести легко генеруються й швидко перевіряються. Для складного коду частка зекономленого часу зменшується, бо головною проблемою стає не написання синтаксично правильного тестового методу, а правильне розуміння всіх сценаріїв поведінки. Це важливе уточнення для команд, які очікують, що генеративні інструменти автоматично вирішать проблему тестування складних модулів.

Результати також показують, що ШІ змінює характер роботи розробника. У ручному підході до підготовки тестів значна частина часу витрачається на безпосереднє написання тестового коду. У підході до підготовки тестів із використанням ШІ ця частина часу скорочується, але зростає роль рев'ю: розробник має перевірити, чи не пропущено важливі гілки, чи коректно сформульовано assertions, чи не розроблено тест, який тільки повторює реалізацію замість перевірки вимог.

Тому застосування засобів ШІ у тестуванні програмного коду потребує не меншої, а іноді й більшої дисципліни перевірки результату.

Окремо потрібно зазначити обмеження дослідження. По-перше, тривалість підготовки тестів оцінювалася за розрахунковою моделлю, а не за прямим хронометражем роботи програмістів. По-друге, AI-assisted тривалість залежить від конкретного інструменту, якості промпта, досвіду розробника та складності предметної області. По-третє, зіставлення production-файлів і тестових файлів за назвами не охоплює всіх можливих способів організації тестів у реальних проєктах. Тому отримані значення потрібно розглядати як порівняльну оцінку тенденції, а не як універсальний норматив часу.

Незважаючи на ці обмеження, загальна тенденція впливу складності розгалужень у програмному коді є методично важливою: branching complexity збільшує трудомісткість процесу тестування C#-коду в обох сценаріях, а засоби ШІ скорочують тривалість переважно за рахунок швидкого розроблення початкового тестового коду. Проте, зі зростанням кількості гілок підвищується ціна помилки, тому людське рев'ю залишається обов'язковим елементом процесу.

Обговорення результатів дослідження. У дослідженні [14] автори проаналізували можливості великих мовних моделей для автоматизованого генерування модульних тестів і встановили, що такі інструменти можуть забезпечувати помітне зростання тестового покриття порівняно з окремими традиційними підходами. Водночас, у роботі наголошено, що ефективність згенерованих тестів залежить від контексту, якості вхідної інформації та здатності моделі враховувати реальну логіку функції. Це узгоджується з результатами цього дослідження, оскільки застосування ШІ справді скорочує тривалість підготовки тестів, однак не усуває потреби у рев'ю, особливо для коду з високою складністю розгалужень C#-коду.

У дослідженні [16] автори дослідили особливості генерування JUnit-тестів за допомогою великих мовних моделей і показали, що результати істотно залежать від набору даних, контексту промпта та складності тестового коду. Зокрема, було виявлено, що згенеровані тести можуть демонструвати високі показники для простіших задач, але значно гірше працювати за умов складніших реальних проєктів. Результати, які отримали ми, підтверджують цю закономірність: для коду з низьким рівнем розгалужень економія часу від використання засобів ШІ є найвищою, тоді як для дуже складного branching вона зменшується через потребу у додатковій перевірці сценаріїв виконання і assertions.

У роботі [17] автори систематизували сучасні напрями використання великих мовних моделей у тестуванні ПЗ та вказали, що LLM можуть бути корисними для генерування тестових випадків, тестових oracle, виявлення помилок і підтримки тестової документації. Водночас, у праці вказано на обмеження підходів, пов'язаних з коректністю, повнотою сценаріїв виконання і надійністю автоматично розроблених тестів. Це підтверджує практичну важливість запропонованої у статті моделі, оскільки вона не розглядає ШІ як повну заміну розробника, а враховує тривалість рев'ю, виправлення та перевірку результатів генерування.

У дослідженні [8] автори запропонували систему автоматизованого генерування та оцінювання тестових

наборів для програмного коду із використанням великих мовних моделей, акцентуючи увагу на переході від простих тестів окремих методів до складніших сценаріїв виконання рівня класів. У цій роботі показано, що наближення до реальних умов розроблення ускладнює оцінювання якості згенерованих тестів і потребує додаткових критеріїв аналізу. Це збігається з результатами нашого дослідження, де зростання branching complexity призводить до збільшення кількості методів тестування, рядків тестового коду та перевірок його гілок, а отже, підвищує вимоги до якості AI-assisted тестування C#-коду.

У дослідженні [11] автори проаналізували спільну еволюцію виробничого та тестового коду у програмних проєктах і показали, що зміни у production-коді часто пов'язані з потребою оновлення або розширення тестів. Такий підхід важливий для нашого дослідження, оскільки підтверджує доцільність аналізу не тільки виробничого коду окремо, а саме зіставлених пар "виробничий файл – тестовий файл". Результати, які отримали ми, деталізують цю ідею через кількісне порівняння складності розгалужень production-коду з кількістю методів тестування, обсягом тестового коду та кількістю assertions.

У роботі [6] автори дослідили зв'язок між тестовим покриттям і характеристиками вихідного коду в open-source системах, звертаючи увагу на те, що покриття тестами не можна інтерпретувати без урахування властивостей самого коду. Це положення є важливим для обговорення отриманих результатів, оскільки наша стаття також виходить із того, що тестові показники залежать не тільки від бажаного рівня якості, а й від структурної складності production-коду. Зокрема, файли з більшою кількістю гілок виконання коду потребують більшого тестового зусилля навіть тоді, коли мають подібний загальний розмір.

У дослідженні [15] автори провели масштабне оцінювання великих мовних моделей для задач unit testing і показали, що такі моделі мають значний потенціал генерування тестів, assertions та інших тестових артефактів. Водночас, у роботі зазначено, що результат залежить від вибраної моделі, способу її налаштування, типу завдання та критеріїв оцінювання якості тестів. Це узгоджується з висновками нашого дослідження: AI-assisted testing може скорочувати тривалість підготовки тестів, але його ефективність зменшується зі зростанням складності розгалужень C#-коду, оскільки у складному коді ключовою стає не швидкість генерування, а правильність і повнота перевірки сценаріїв виконання.

Обговорення наявних результатів дослідження у контексті сучасних потреб IT-галузі підтверджує доцільність проведення саме цього дослідження. У науковій літературі вже розглянуто автоматизоване генерування тестів, можливості великих мовних моделей, тестове покриття та зв'язок між production-кодом і тестовим кодом, однак недостатньо висвітлено практичне питання кількісного порівняння тривалості ручного та AI-assisted підготовки тестів для різних рівнів складності розгалужень C#-коду. Запропонований підхід доповнює наявні роботи тим, що поєднує структурні показники branching complexity з оцінюванням трудомісткості процесу тестування C#-коду та показує, як змінюється ефективність використання засобів III залежно від складності виробничого коду.

Отже, внаслідок виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – розроблено методiku оцінювання трудомісткості процесу тестування програмного коду у спосіб поєднання структурних показників складності розгалужень C#-коду з розрахунковими моделями тривалості ручного та AI-assisted підготовки тестів, що дало змогу встановити залежність між рівнем branching complexity, обсягом тестового коду, кількістю assertions і зміною ефективності використання інструментів штучного інтелекту.

Практична значущість результатів дослідження – можна використати для планування QA-робіт, оцінювання трудомісткості підготовки тестів, визначення ділянок коду з підвищеним ризиком недостатнього його тестування, пріоритезації рефакторингу C#-коду та обґрунтованого вибору його фрагментів, для яких застосування AI-assisted testing є найдоцільнішим як допоміжний інструмент розробника.

Висновки / Conclusions

Розроблено моделі тривалості підготовки тестів вручну і засобами III, які дали можливість обґрунтувати доцільність застосування показників branching complexity під час оцінювання QA-завдань, пріоритезації рефакторингу програмного коду та вибору його ділянок для AI-assisted testing. За результатами проведення дослідження сформульовано такі висновки.

1. Систематизовано підходи до оцінювання складності розгалужень програмного коду та обґрунтовано доцільність використання показників branching complexity для аналізу трудомісткості процесу тестування C#-коду, оскільки кількість умов, циклів, вкладених блоків і альтернативних шляхів його виконання безпосередньо впливає на кількість сценаріїв їх виконання, які потрібно перевірити.
2. Проаналізовано зіставлені пари виробничих і тестових файлів C#-проєктів та з'ясовано, що складність розгалужень C#-коду впливає на кількість методів тестування, обсяг тестового коду, кількість assertions і загальну тривалість підготовки тестів.
3. Встановлено, що зі зростанням складності розгалужень C#-коду кількість методів тестування збільшується від 4,0 до 23,5, обсяг тестового коду – від 41,6 до 452,2 рядка, а кількість assertions – від 4,8 до 71,3, що підтверджує потребу в детальнішій перевірці коду з більшою кількістю альтернативних сценаріїв його виконання.
4. Виконано розрахунок тривалості ручної підготовки тестів і встановлено, що перехід від низького до дуже високого рівня складності розгалужень C#-коду збільшує трудомісткість приблизно від 20,3 до 160,9 хв, тобто майже у 8 разів, тому цей показник доцільно враховувати під час планування QA-завдань, оцінювання технічного боргу та визначення пріоритетів рефакторингу програмного коду.
5. Застосовано модель підготовки тестів із використанням III та з'ясовано, що використання інструментів III скорочує тривалість підготовки тестів у всіх групах складності C#-коду, однак цей ефект зменшується зі зростанням рівня його розгалуженості: для простого коду економія часу становить близько 60 %, а для дуже складного – близько 35 %.
6. Встановлено, що III є корисним допоміжним інструментом для розроблення початкових варіантів тестів, генерування типових сценаріїв виконання і пришвидшення рутинного кодування, однак не замінює ревіз

розробника, перевірку граничних випадків і коректності assertions, особливо для коду з високим рівнем його розгалуженості.

Декларація про використання штучного інтелекту. Під час підготовки цієї роботи автори використовували сервіси ChatGPT та Grammarly для: перевірки граматики та орфографії, перефразування та переформулювання складних речень з текстовими зворотами. Після використання цих сервісів автори проаналізували та відредагували контент майбутньої статті та несуть повну відповідальність за зміст публікації.

References

1. Ammann, P., & Offutt, J. (2016). Introduction to software testing (2nd ed.). Cambridge University Press. URL: <https://www.cambridge.org/highereducation/books/introduction-to-software-testing/9D17AF6FA69D6C97E7AAB1239AFA5BAE>
2. Andrzejewski, M., Dubicka, N., Podolak, J., Kowal, M., & Siłka, J. (2025). Automated test generation using large language models. *Data*, 10(10), article number 156. <https://doi.org/10.3390/data10100156>
3. Daka, E., & Fraser, G. (2014). A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering* (pp. 201–211). IEEE. <https://doi.org/10.1109/ISSRE.2014.11>
4. Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. (2023). GitHub Copilot AI pair programmer: Asset or liability? *Journal of Systems and Software*, 203, article ID 111734. <https://doi.org/10.1016/j.jss.2023.111734>
5. Fraser, G., & Arcuri, A. (2011). EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (pp. 416–419). Association for Computing Machinery. <https://doi.org/10.1145/2025113.2025179>
6. Imran, M., Shang, W., & Hassan, A. E. (2025). Is code coverage of performance tests related to source code features? An empirical study on open-source Java systems. *Empirical Software Engineering*. <https://doi.org/10.1007/s10664-025-10712-3>
7. International Organization for Standardization. (2023). ISO/IEC 25010:2023 Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – Product quality model. URL: <https://www.iso.org/standard/78176.html>
8. Lops, A., Narducci, F., Ragone, A., Trizio, M., & Bartolini, C. (2024). A system for automated unit test generation using large language models and assessment of generated test suites. *arXiv*. <https://doi.org/10.48550/arXiv.2408.07846>
9. Martin, R. C. (2008). Clean code: A handbook of agile software craftsmanship. *Prentice Hall*. URL: <https://www.pearson.de/clean-code-a-handbook-of-agile-software-craftsmanship-9780132350884>
10. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), 308–320. <https://doi.org/10.1109/TSE.1976.233837>
11. Miranda, C., Zaidman, A., & Panichella, A. (2025). Test co-evolution in software projects: A large-scale empirical study. *Journal of Software: Evolution and Process*. <https://doi.org/10.1002/smr.70035>
12. Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing* (3rd ed.). John Wiley & Sons. URL: <https://www.wiley.com/en-us/The+Art+of+Software+Testing%2C+3rd+Edition-p-9781118031964>
13. Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). *The impact of AI on developer productivity: Evidence from GitHub Copilot*. *arXiv*. <https://doi.org/10.48550/arXiv.2302.06590>
14. Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2024). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1), 85–105. <https://doi.org/10.1109/TSE.2023.3334955>
15. Shang, Y., Zhang, Q., Fang, C., Gu, S., Zhou, J., & Chen, Z. (2024). A large-scale empirical study on fine-tuning large language models for unit testing. *arXiv*. <https://doi.org/10.48550/arXiv.2412.16620>
16. Siddiq, M. L., Santos, J. C. S., Tanvir, R. H., Ulfat, N., Al Rifat, F., & Lopes, V. C. (2023). Using large language models to generate JUnit tests: An empirical study. *arXiv*. <https://doi.org/10.48550/arXiv.2305.00418>
17. Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4), 911–936. <https://doi.org/10.1109/TSE.2024.3368208>
18. Wei, Y., Meyer, B., & Oriol, M. (2012). Is branch coverage a good measure of testing effectiveness? In *Empirical software engineering and verification* (pp. 194–212). Springer. https://doi.org/10.1007/978-3-642-25231-0_5

M. M. Seniv, V. V. Inshyn

Lviv Polytechnic National University, Lviv, Ukraine

ASSESSING THE IMPACT OF C# BRANCHING COMPLEXITY ON TEST WRITING TIME MANUALLY AND WITH AI ASSISTANCE

Modern software projects practical need has been identified for quantitative assessment of how the complexity of program code branches affects the laboriousness of creating tests in C# projects. It has been established that for such an assessment it is not enough to take into account only the file size or the fact of the presence of tests, since the real costs of testing depend on the number of alternative execution paths, the depth of nesting, the number of test methods, the volume of test code and the number of checks required. It has been found that the complexity of branches is directly related to the number of scenarios that need to be considered during unit testing: each additional condition, loop or nested block increases the number of situations for which it is necessary to select input data, determine the expected result and formulate a check. The time for writing tests has been estimated using two approaches: manual writing and creating tests using artificial intelligence tools. A model of the manual approach has been characterized, in which the time depends on the number of test methods, lines of test code and the cost of preparing one test. A model of AI usage is described, which additionally takes into account the time for query formulation, context clarification, test code generation, review, edge case finding, and correction of incorrect checks. It is found that the transition from a low to a very high level of branching is accompanied by an increase in the estimated time for manual test writing from approximately 20 to 161 minutes, i.e. almost eight times. It is found that the use of AI reduces time in all complexity groups, but this effect decreases with increasing code complexity: for simple code, the savings are about 60 %, and for very complex code, about 35 %. It is found that the reason for this reduction is the transfer of part of the work from writing tests to checking logic, correctness of assertions, compliance with business rules, and completeness of scenarios. The practical value of the work is that branch complexity metrics can be used for testing planning, evaluating QA tasks, prioritizing refactoring, and identifying areas of code where AI is a useful auxiliary tool, but cannot replace expert developer review.

Keywords: complexity of C# code branching; test scenarios for C# code; test checks; refactoring of programme code; the labour-intensity of test preparation; modular testing of C# code.