МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: Розробка та оцінка ефективності автоматизованого тестування за допомогою PyTest

Здобувача вищої освіти другого (магістерського) рівня,
II курсу, групи КМІ-24-2, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології в економіці»


(підпис)

Лопатка Дениса Вікторовича
(прізвище, ім'я, по батькові)

Науковий керівник


(підпис)

Грищенко Світлана Миколаївна, канд. пед. наук, ст. досл.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми


(підпис)

Омельчук Антон Анатолійович, канд. техн. наук
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри протокол від «02» січня 2026 № 10

Завідувач кафедри


(підпис)

Філоненко Михайло Миколайович, канд. фіз.-мат. наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2026

Реєстрація №25 22.01.2026 _____
(дата) (підпис відповідальної особи)

Результати перевірки до захисту 02.01.26р _____
(до захисту, на доопрацювання) (дата) (підпис)

Результат захисту 5 (відмінно) 05.02.2026 _____
(оцінка) (дата) (підпис)

Голова комісії: _____ О.С. Дюбанов
(підпис)

Члени комісії:

_____ А.Ю. Горбовий
(підпис)

_____ М.М. Філоненко
(підпис)

_____ І.Д. Погорєловська
(підпис)

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст кваліфікаційної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	28		05.02.2026 Дюбанов О.С.
2. Дотримання вимог щодо оформлення	від 0 до 10	9		Горбовий А.Ю.
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	5		Філоненко М.М. Погорєловська І.Д.
4. Рецензент	від 0 до 5	5		Суховецька Л.П.
РАЗОМ БАЛІВ	від 0 до 50	47		

Оцінювання кваліфікаційної роботи науковим керівником - 45 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
92	A	5	05.02.2026	Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Інформаційні управляючі системи і технології в економіці»

Затверджую
Завідувач кафедри _____ Філоненко М.М.
«11» грудня 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Лопатці Денису Вікторовичу
Тема кваліфікаційної роботи Розробка та оцінка ефективності автоматизованого тестування за допомогою PyTest
Затверджена наказом ректора від «10» грудня 2024 № 1852
1. Строк здачі закінченої роботи «02» січня 2026
2. Мета роботи Розробити та експериментально оцінити підхід до автоматизованого тестування на базі PyTest для нефронтендних застосуванням інструментів NLP (LanguageTool, spaCy) та засобів звітності.
Об'єкт дослідження Процес забезпечення якості нефронтендних програмних систем шляхом автоматизованого тестування.
Предмет дослідження Методи автоматизованого тестування на базі PyTest.
Орієнтовна база дослідження Тестовий проєкт у середовищі Windows, корпус тестових документів та інструменти аналізу тексту LanguageTool і spaCy.
3. Вихідні дані до роботи Вихідні дані до роботи вимоги до обсягу та структури; корпус тестових документів, програмний стек Python + PyTest; інструменти LanguageTool та spaCy; плагіни pytest-html, pytest-cov, pytest-xdist;
4. Перелік графічного матеріалу Життєвий цикл програмного забезпечення; Життєвий цикл автоматизації тестування; Порівняння PyTest з іншими фреймворками тестування; Узагальнені підходи до тестування нефронтендних систем; Архітектура моделі автоматизованого тестування на базі PyTest; Структура тестового проєкту та набір тестових сценаріїв; Потік даних "документ → аналіз → тестове рішення → звіт";
5. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)
Розділ 1. Теоретичні та методологічні засади автоматизованого тестування програмного забезпечення. Поняття та класифікація автоматизованого тестування. Фреймворк pytest та його місце серед інструментів автоматизованого тестування. Розділ 2. Аналіз та проєктування підходу до автоматизованого тестування десктопних та інтелектуальних програмних систем. Аналіз предметної області та існуючих підходів до тестування нефронтендних систем. Проєктування моделі автоматизованого тестування на базі pytest. Розділ 3. Практична реалізація автоматизованого тестування нефронтендних програмних систем на базі pytest. Характеристика об'єкта та середовища практичної реалізації. Проєктування та реалізація автоматизованих тестових сценаріїв.

7. Календар план виконання роботи:

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	01.12.2024-15.12.2024	01.12.2024-17.12.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	16.12.2024-30.05.2025	17.12.2024-04.06.2025
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	31.05.2025-10.09.2025	04.06.2025-13.09.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	11.09.2025-20.11.2025	13.09.2025-25.11.2025
5	**Висновки**	21.11.2025-11.12.2025	25.11.2025-11.12.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	12.12.2025-25.12.2025	12.12.2025-25.12.2025
7	Отримання відгуку керівника	26.12.2025	26.12.2025
8	Отримання зовнішньої рецензії	10.01.2026	10.01.2026

8. Дата видачі завдання «11» грудня 2024

9. Керівник кваліфікаційної роботи _____ (підпис) Грищенко С. М. (прізвище, ініціали)

10. Керівник освітньо-професійної програми _____ (підпис) Омельчук А.А. (прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти _____ (підпис) Лопатка Д. В. (прізвище, ініціали)

АНОТАЦІЯ

Розробка та оцінка ефективності автоматизованого тестування за допомогою PyTest

(назва кваліфікаційної роботи)

Мета роботи полягає в розробці та оцінюванні ефективності автоматизованого тестування на базі PyTest для нефронтендних програмних систем із застосуванням інструментів аналізу природної мови для контролю якості текстового контенту.

Завданнями роботи є: проаналізувати існуючі підходи до автоматизації тестування; обґрунтувати використання PyTest; реалізувати конвеєр аналізу TXT/PDF-документів; визначити правила класифікації помилок за критичністю та порогові критерії прийняття тестових рішень; сформувати показники для оцінювання ефективності.

Предметом дослідження є засоби PyTest та методи інтерпретації результатів лінгвістичного аналізу в тестових сценаріях. Об'єктом дослідження виступає процес забезпечення якості нефронтендних програмних систем шляхом автоматизованого тестування. Базою дослідження виступає тестовий проєкт у середовищі Windows (Python, PyTest), корпус документів (TXT, PDF) та інструменти LanguageTool і spaCy.

За результатами дослідження розроблено відтворюваний підхід до автоматизованого контролю якості тексту та визначено критерії ефективності тестування. Одержані результати можуть бути використані для регресійних перевірок контенту та інтеграції тестів у CI.

Кваліфікаційна робота містить 43 сторінки, 8 таблиць, 11 рисунків, список використаних джерел з 40 найменувань, 7 додатків. Ключові слова: PyTest, автоматизоване тестування, Windows, нефронтендні системи, NLP, LanguageTool, spaCy, критичність помилок, порогові критерії.

Рік виконання кваліфікаційної роботи: 2025. Рік захисту роботи: 2026.

ABSTRACT

Development and Evaluation of the Effectiveness of Automated Testing Using PyTest

(title of the qualification thesis)

The aim of this thesis is to develop and evaluate the effectiveness of automated testing based on PyTest for non-frontend software systems, using natural language processing tools to ensure the quality of textual content.

The objectives of the thesis are to: analyze existing approaches to test automation; justify the use of PyTest; implement a pipeline for analyzing TXT/PDF documents; define rules for classifying errors by severity and threshold criteria for making test decisions; and establish metrics for evaluating effectiveness.

The subject of the research is PyTest tools and methods for interpreting linguistic analysis results within test scenarios. The object of the research is the quality assurance process for non-frontend software systems through automated testing. The research base includes a test project in a Windows environment (Python, PyTest), a corpus of documents (TXT, PDF), and the LanguageTool and spaCy tools.

As a result, a reproducible approach to automated text quality control was developed and testing effectiveness criteria were defined. The obtained results can be used for content regression checks and for integrating tests into CI pipelines.

The qualification thesis contains 43 pages, 8 tables, 11 figures, a list of references with 40 entries, and 7 appendices. Keywords: PyTest, automated testing, Windows, non-frontend systems, NLP, LanguageTool, spaCy, error severity, threshold criteria.

Year of thesis completion: 2025. Year of defense: 2026.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 3

ВСТУП 4

РОЗДІЛ 1. ТЕОРЕТИЧНІ ТА МЕТОДОЛОГІЧНІ ЗАСАДИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 7

1.1. Поняття та класифікація автоматизованого тестування 7

1.2. Фреймворк pytest та його місце серед інструментів автоматизованого тестування 10

РОЗДІЛ 2. АНАЛІЗ ТА ПРОЄКТУВАННЯ ПІДХОДУ ДО АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ДЕСКТОПНИХ ТА ІНТЕЛЕКТУАЛЬНИХ ПРОГРАМНИХ СИСТЕМ 19

2.1. Аналіз предметної області та існуючих підходів до тестування нефронтендних систем 19

2.2. Проєктування моделі автоматизованого тестування на базі pytest 23

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НЕФРОНТЕНДНИХ ПРОГРАМНИХ СИСТЕМ НА БАЗІ PYTEST 31

3.1. Характеристика об’єкта та середовища практичної реалізації 31

3.2. Проєктування та реалізація автоматизованих тестових сценаріїв ... 33

ВИСНОВКИ 43

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 46

ДОДАТКИ 49

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CI — Continuous Integration, безперервна інтеграція

CLI — Command Line Interface, інтерфейс командного рядка

E2E — End-to-End testing, наскрізне тестування

GUI — Graphical User Interface, графічний інтерфейс користувача

NLP — Natural Language Processing, обробка природної мови

OS — Operating System, операційна система

PDF — Portable Document Format, формат електронних документів

QA — Quality Assurance, забезпечення якості

SDLC — Software Development Life Cycle, життєвий цикл розроблення ПЗ

SUT — System Under Test, система, що тестується

UI — User Interface, інтерфейс користувача

JSON — JavaScript Object Notation, текстовий формат обміну даними

API — Application Programming Interface, програмний інтерфейс взаємодії застосунків

VENV — Virtual Environment, ізольоване середовище Python

NLP-модель — попередньо навчена модель для задач обробки природної мови (у роботі — модель spaCy)

ВСТУП

Сучасні інформаційні системи характеризуються високою складністю, швидкими темпами розвитку та постійними змінами функціоналу. За таких умов критично зростає потреба у забезпеченні стабільності, надійності та якості програмного забезпечення. Одним із ключових інструментів досягнення цієї мети є автоматизоване тестування, яке дозволяє своєчасно виявляти дефекти, знижувати ризики помилок та підвищувати якість програмних продуктів.

Особливої актуальності набуває застосування автоматизованого тестування в умовах безперервної інтеграції та доставки програмного забезпечення, де ручне тестування є недостатньо ефективним. Фреймворк pytest є одним із сучасних і широко застосовуваних інструментів автоматизованого тестування мовою Python, який забезпечує гнучкість, масштабованість і високу швидкість виконання тестів. Разом із цим, питання методично обґрунтованої побудови автотестів, структурування тестових сценаріїв та інтеграції автоматизованого тестування у процес розробки залишаються недостатньо систематизованими.

У зв’язку з цим дослідження та практична реалізація автоматизованого тестування із застосуванням pytest є актуальним науково-практичним завданням.

Проблематику автоматизованого тестування програмного забезпечення досліджували як вітчизняні, так і зарубіжні науковці та практики у сфері програмної інженерії. У наукових і прикладних публікаціях розглядаються загальні принципи тестування, підходи до побудови тестових сценаріїв, використання unit-, integration- та end-to-end тестів. Значна увага приділяється автоматизації тестування з використанням Python та відповідних фреймворків.

Разом із тим, більшість наявних робіт зосереджена на загальних підходах до тестування або описі окремих інструментів без детального аналізу їх практичного застосування в реальних проєктах. Недостатньо розкритими залишаються питання організації структури автотестів, побудови масштабованої тестової архітектури та ефективного використання можливостей pytest у прикладних інформаційних системах.

Метою кваліфікаційної роботи є дослідження та реалізація автоматизованого тестування програмного забезпечення із застосуванням фреймворку pytest для підвищення якості та надійності інформаційних систем.

Для досягнення поставленої мети в роботі необхідно вирішити такі завдання:

- проаналізувати теоретичні засади автоматизованого тестування програмного забезпечення;
- дослідити сучасні підходи та інструменти автоматизованого тестування;
- обґрунтувати вибір фреймворку pytest для реалізації автотестів;
- розробити структуру автоматизованих тестових сценаріїв;
- реалізувати прикладні автотести із застосуванням pytest;
- оцінити ефективність використання автоматизованого тестування у процесі перевірки програмного забезпечення. Об’єктом дослідження є процес тестування програмного забезпечення в інформаційних системах.

Предметом дослідження є методи та засоби автоматизованого тестування програмного забезпечення з використанням фреймворку pytest мовою Python.

У роботі використано загальнонаукові та спеціальні методи дослідження, зокрема: аналіз і синтез — для вивчення теоретичних основ тестування; порівняльний аналіз — для оцінки існуючих інструментів автоматизованого тестування; моделювання та проєктування — для побудови структури автотестів; практичне програмування — для реалізації автоматизованих тестових сценаріїв; експериментальний метод — для перевірки працездатності та ефективності розроблених рішень.

Базою дослідження є програмні модулі та прикладні програмні компоненти інформаційних систем, для яких здійснюється автоматизоване тестування із використанням мови програмування Python та фреймворку pytest.

Практична значимість роботи полягає у можливості використання розроблених підходів та автоматизованих тестових сценаріїв у процесі тестування програмного забезпечення з метою підвищення його якості, надійності та зменшення витрат часу на перевірку функціоналу. Отримані

результати можуть бути використані під час розробки та супроводу інформаційних систем, а також у навчальному процесі при вивченні дисциплін, пов’язаних із тестуванням програмного забезпечення.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ТА МЕТОДОЛОГІЧНІ ЗАСАДИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Поняття та класифікація автоматизованого тестування

Автоматизоване тестування програмного забезпечення трактується як формалізований процес перевірки коректності функціонування програмних компонентів із застосуванням програмних засобів, що виконують тестові сценарії без безпосередньої участі людини[23]. На відміну від ручного тестування, автоматизовані тести характеризуються відтворюваністю, масштабованістю та можливістю інтеграції у безперервні процеси розробки.

За функціональним призначенням автоматизоване тестування поділяється на функціональне та нефункціональне. Функціональне тестування спрямоване на перевірку відповідності програмного забезпечення визначеним вимогам, тоді як нефункціональне охоплює перевірку продуктивності, надійності, безпеки та якості обробки даних[25].

Залежно від рівня абстракції виділяють модульне, інтеграційне, системне та регресійне тестування. Модульне тестування зосереджується на окремих компонентах програмної системи і є базовим рівнем автоматизації. Інтеграційне тестування дозволяє оцінити коректність взаємодії між модулями, а регресійне тестування забезпечує контроль стабільності системи після внесення змін[1].

Доцільно наочно представити життєвий цикл програмного забезпечення у вигляді узагальненої схеми (див.рис.1.1).

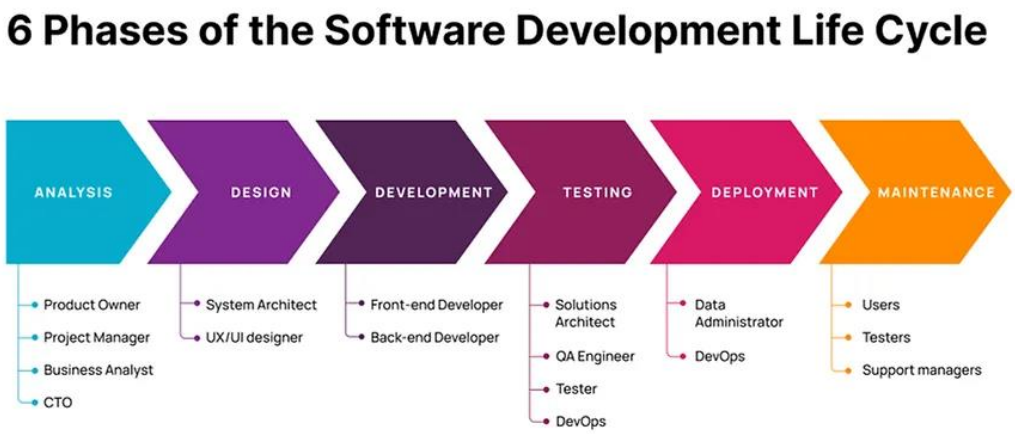


Рис. 1.1. Життєвий цикл програмного забезпечення.

Джерело: [онлайн видавництво Medium]

Доцільно наочно представити класифікацію автоматизованого тестування у вигляді узагальненої схеми (див.рис.1.2.).



Рис. 1.2 Схема життєвого циклу автоматизації тестування

Джерело: [авторський рисунок]

У наукових працях з інженерії програмного забезпечення автоматизоване тестування розглядається як один із ключових механізмів забезпечення якості програмних систем. Згідно з підходами, викладеними у фундаментальних

роботах з тестування, раннє виявлення дефектів дозволяє суттєво знизити вартість їх усунення на наступних етапах життєвого циклу програмного забезпечення[14].

Як зазначає Г. Майєрс, ефективне тестування не полягає у доведенні коректності роботи програмного продукту, а спрямоване на систематичне виявлення помилок і слабких місць реалізації[17]. Даний підхід залишається актуальним і в умовах сучасних гнучких методологій розробки, де автоматизоване тестування інтегрується у процеси безперервної інтеграції.

У працях Р. Прессмана автоматизоване тестування розглядається як невід'ємна складова інженерних процесів, що забезпечує контроль якості як на рівні окремих модулів, так і на рівні інтегрованих систем. Автор підкреслює важливість використання формалізованих тестових сценаріїв для досягнення відтворюваних результатів.

Міжнародні стандарти IEEE також визначають автоматизоване тестування як формалізований процес перевірки відповідності програмного забезпечення встановленим вимогам і очікуванням користувачів. Застосування стандартів дозволяє уніфікувати підходи до тестування та забезпечити прозорість оцінювання якості програмних продуктів.

Таким чином, аналіз наукових джерел підтверджує доцільність використання автоматизованого тестування як базового інструменту контролю якості програмного забезпечення, що створює методологічне підґрунтя для подальшого дослідження можливостей фреймворку pytest.

У сучасній інженерії програмного забезпечення автоматизоване тестування розглядається як один із ключових механізмів забезпечення якості програмних систем на всіх етапах їх життєвого циклу. Зі зростанням складності програмних продуктів, збільшенням обсягу функціональних вимог та скороченням термінів розробки традиційні підходи до ручного тестування втрачають ефективність. Це зумовлює необхідність використання формалізованих, відтворюваних і масштабованих методів перевірки коректності програмного забезпечення.

Автоматизоване тестування дозволяє здійснювати регулярний контроль якості програмних компонентів, зменшувати вплив людського фактора та забезпечувати стабільність результатів перевірок. Особливого значення воно набуває в умовах ітеративних та гнучких методологій розробки, де програмний код постійно змінюється, а регресійні помилки можуть виникати на будь-якому етапі розвитку проєкту. У такому контексті автоматизовані тести виступають не лише інструментом виявлення дефектів, а й засобом підтримки керованості процесу розробки.

Водночас характер сучасних програмних систем поступово змінюється: поряд із вебзастосунками все більшу роль відіграють десктопні рішення, сервіси обробки даних, аналітичні модулі та системи з елементами штучного інтелекту. Це ставить нові вимоги до інструментів автоматизованого тестування, які мають бути достатньо універсальними, щоб застосовуватися поза межами класичних вебінтерфейсів.

1.2. Фреймворк pytest та його місце серед інструментів автоматизованого тестування

На практиці автоматизоване тестування реалізується за допомогою різноманітних інструментів, які умовно можна поділити за орієнтацією на тип програмної системи. Значна частина сучасних фреймворків, таких як Selenium, Cypress або Playwright, орієнтована передусім на автоматизацію тестування вебінтерфейсів. Вони забезпечують ефективну перевірку взаємодії користувача з браузером, однак їх застосування для тестування нефронтендних компонентів є обмеженим або надмірно складним.

У практиці автоматизації тестування вебінтерфейсів Selenium тривалий час залишається де-факто стандартом завдяки підтримці різних мов програмування, можливості керування основними браузерами та широкій інтегрованості у CI-процеси. Його сильна сторона — універсальність, однак саме ця універсальність часто “платиться” підвищеною чутливістю до нестабільності середовища: час очікувань, синхронізація, завантаження сторінок і залежність

від стану мережі формують типові джерела “flaky”-поведінки UI-тестів. У контексті оцінки ефективності автоматизованого тестування це критично, оскільки такі тести можуть погіршувати головний показник — надійність сигналу про регресію, коли падіння не обов’язково означає дефект у коді. З цієї причини Selenium доцільно розглядати як інструмент для end-to-end перевірок верхнього рівня, але не як основу вимірюваної ефективності тестування бізнес-логіки[27].

Cypress та Playwright репрезентують “нову хвилю” інструментів UI-автоматизації, що намагаються зменшити проблему нестабільності за рахунок більш керованого середовища виконання та автоматичних механізмів очікування. Cypress орієнтований на тісну інтеграцію з браузером і надає сильні засоби налагодження, однак фактично прив’язує тестову інфраструктуру до JavaScript-екосистеми і до специфічної моделі виконання тестів у браузерному середовищі. Це робить його менш зручним для проєктів, де основний контроль якості організовано в Python-стеку або де частина перевірок не є браузерними. Playwright, у свою чергу, забезпечує кросбраузерність і розширені можливості автоматизації, але залишається інструментом UI-рівня: він підсилює стабільність взаємодії з інтерфейсом, однак не усуває фундаментальної проблеми — UI-тести перевіряють наслідки роботи системи, а не її причини, і тому мають вищу вартість підтримки при зміні інтерфейсу[22; 25].

З позиції теми «Розробка та оцінка ефективності автоматизованого тестування за допомогою PyTest» важливо розділяти дві площини: автоматизацію як факт (наявність автотестів) і ефективність автоматизації як властивість (якість сигналу, швидкість зворотного зв’язку, діагностичність, супроводжуваність). Для досягнення вимірюваної ефективності базовою повинна бути автоматизація на рівні модулів і компонентів, де дефект локалізується швидше, а повторюваність вища. Тому у даній роботі UI-інструменти розглядаються як можливе доповнення для димових end-to-end перевірок, тоді як ядром автоматизації виступає модульний фреймворк pytest,

який дозволяє будувати контроль якості на рівні бізнес-логіки й обробки даних[24].

Для перевірки логіки програмних компонентів на рівні коду використовуються модульні фреймворки, зокрема unittest і pytest для мови Python. Такі інструменти дозволяють ізолювати окремі функції та класи, що є особливо важливим для систем без графічного інтерфейсу. Серед них pytest вирізняється більш гнучкою архітектурою, підтримкою параметризації тестів і розвиненою системою фікстур, що спрощує підготовку тестового середовища.

Попри наявність великої кількості інструментів, більшість з них орієнтовані на тестування детермінованих сценаріїв з чітко визначеними очікуваними результатами. Це створює труднощі при застосуванні автоматизованого тестування до систем, результати роботи яких не завжди є однозначними, зокрема до систем аналізу даних або текстового контенту.

Фреймворк pytest є одним із найбільш поширених інструментів автоматизованого тестування програмного забезпечення, розробленого мовою Python. Його популярність зумовлена поєднанням простоти використання, високої гнучкості та широких можливостей розширення. На відміну від класичних підходів до тестування, pytest не вимагає створення складних ієрархій класів або використання шаблонного коду, що суттєво знижує поріг входу для розробників і тестувальників.

У офіційній документації pytest підкреслюється, що фреймворк орієнтований на підтримку модульного тестування та інтеграцію з сучасними інструментами розробки програмного забезпечення, зокрема системами контролю версій і безперервної інтеграції [27]. Завдяки цьому pytest широко використовується не лише в навчальних проєктах, але й у промислових програмних системах.

Ключовим елементом архітектури pytest є механізм фікстур, який дозволяє формалізувати підготовку тестового середовища та керувати життєвим циклом тестових даних. Використання фікстур забезпечує ізоляцію тестів, повторне використання коду та підвищує надійність результатів тестування. Подібний

підхід відповідає рекомендаціям, описаним у роботах з тестових шаблонів та практик xUnit [28].

У порівнянні з вбудованим модулем unittest, pytest забезпечує більш декларативний стиль опису тестів, підтримує параметризацію та дозволяє легко розширювати функціональність за допомогою плагінів. Це робить pytest придатним для використання у проєктах з різною архітектурою та масштабом, включаючи нефронтендні програмні системи.

Додатковою відмінністю pytest у порівнянні з мінімалістичними модульними фреймворками є підтримка масштабування тестових прогонів без зміни тестового коду. Практично це означає, що при зростанні обсягу тестового набору або при необхідності регулярних регресійних прогонів ефективність автоматизації залежить від можливості виконувати тести паралельно, швидко і відтворювано. Для таких задач у pytest застосовуються розширення на кшталт pytest-xdist, що дозволяють розподіляти виконання тестів між процесами та скорочувати час “time-to-feedback”. В умовах, коли результат автоматизації використовується як контрольна точка в CI, скорочення часу виконання тестів прямо впливає на ефективність процесу розробки, оскільки зменшує затримку між внесенням змін і отриманням сигналу про дефект[26].

Ще одним напрямом підсилення аналітичної цінності тестів є використання підходів, що орієнтовані на генерацію даних і перевірку властивостей (property-based testing). У таких підходах тест перевіряє не конкретний приклад “вхід-вихід”, а інваріанти та властивості функції на широкому спектрі даних. Для Python це реалізується, зокрема, через бібліотеку Hypothesis, яка природно поєднується з pytest. У межах даної роботи це важливо для тестів, де потрібна оцінка стабільності алгоритмів обробки тексту або перевірка того, що певні трансформації не змінюють ключові властивості результату. В таких задачах ефективність автоматизації підвищується через збільшення “простору” перевірок без пропорційного збільшення обсягу вручну написаних тестів [48].

Таким чином, аналіз функціональних можливостей pytest свідчить про доцільність його використання як універсального фреймворку автоматизованого тестування, здатного підтримувати перевірку як серверних, так і десктопних програмних компонентів

Для узагальнення ключових відмінностей pytest доцільно подати порівняльну схему з іншими фреймворками тестування (див.рис.1.3.).

Критерій	**pytest**	**unittest**	**nose2**
Поріг входу	+	±	±
Фікстури/fixtures	+	−	±
Параметризація	+	±	±
Екосистема плагінів	+	−	±
Звіти/репорти	+	±	±

Рис. 1.3. Порівняння pytest з іншими фреймворками тестування

Джерело: [авторський рисунок]

Перспективи використання pytest для нефронтендних програмних систем.

Нефронтендні програмні системи охоплюють широкий спектр рішень, включаючи десктопні застосунки, фонові сервіси, утиліти обробки даних та аналітичні модулі. Для таких систем характерною є відсутність або другорядна роль користувацького інтерфейсу, що знижує ефективність UI-орієнтованих підходів до тестування.

Однією з основних проблем автоматизованого тестування нефронтендних систем є необхідність перевірки бізнес-логіки та алгоритмічних процесів, а не зовнішньої поведінки інтерфейсу. У таких умовах ключовим стає тестування коректності обробки даних, виконання правил предметної області та стабільності функціонування внутрішніх компонентів.

Для нефронтендних систем найбільш характерним є змішаний профіль тестування: з одного боку, існує потреба перевіряти кінцеву поведінку через зовнішні ознаки (наприклад, реакцію інтерфейсу десктопного застосунку), а з іншого — саме внутрішня логіка визначає правильність роботи системи. На практиці десктоп-автоматизація у Windows часто реалізується через інструменти, що працюють поверх Windows UI Automation або подібних API. Прикладами є WinAppDriver або бібліотеки на кшталт pywinauto, які дозволяють програмно знаходити елементи інтерфейсу, виконувати дії та верифікувати базові сценарії “smoke/regression” на рівні UI[32; 33]. Однак такі тести мають типові ризики нестабільності: залежність від стану ОС, роздільної здатності, фокусу вікон, таймінгів і фонового навантаження.

З огляду на це, у літературі та практиці тестування закріпився підхід, у якому UI-автоматизація використовується обмежено — як контроль ключових користувацьких потоків, тоді як основний обсяг автоматизації переноситься на рівень модулів і сервісів, де тести дешевші у підтримці та надійніші. Для теми даної роботи це є принциповим: оцінювати ефективність автоматизованого тестування доцільно на тому шарі, де отримується найбільш стабільний сигнал, а саме на рівні логіки обробки даних. Саме тут pytest дає можливість побудувати системну перевірку, не залежну від UI, але сумісну з десктопним середовищем, оскільки запуск і ізоляція тестів не прив’язані до браузера чи DOM-структури[34; 35].

Особливої складності набуває тестування систем, що працюють з текстовими або іншими неструктурованими даними. Алгоритми аналізу природної мови, автоматичного виправлення помилок або класифікації тексту часто мають імовірнісний характер, що ускладнює формалізацію критеріїв успішності тестів. Це потребує використання адаптивних підходів до оцінювання результатів, які виходять за межі простого порівняння очікуваних і фактичних значень.

Розвиток сучасних програмних систем супроводжується зростанням кількості нефронтендних рішень, які функціонують без вебінтерфейсу або мають

мінімальну взаємодію з користувачем. До таких систем належать десктопні застосунки, фонові сервіси, утиліти обробки даних, аналітичні модулі та програмні компоненти з елементами штучного інтелекту. Забезпечення якості таких систем потребує інструментів тестування, орієнтованих насамперед на перевірку логіки обробки даних і стабільності алгоритмів[40].

Традиційні підходи до автоматизації тестування нефронтендних систем часто зосереджуються на перевірці графічного інтерфейсу користувача. Однак UI-орієнтовані тести характеризуються високою чутливістю до змін інтерфейсу, складністю супроводу та низькою відтворюваністю. У наукових і прикладних роботах підкреслюється доцільність переходу до тестування внутрішньої логіки програмних систем як більш стабільного та масштабованого підходу[36].

Фреймворк pytest завдяки своїй архітектурі добре пристосований для реалізації такого підходу. Використання модульних тестів і фікстур дозволяє здійснювати перевірку нефронтендних компонентів незалежно від способу взаємодії з користувачем. Це створює передумови для використання pytest у тестуванні десктопних застосунків у середовищі Windows, де основний акцент робиться на коректності бізнес-логіки та обробки даних.

Окремим напрямом застосування pytest є тестування програмних систем, що працюють з неструктурованими текстовими даними. У таких системах результати обробки можуть мати ймовірнісний характер, що ускладнює використання класичних критеріїв успішності тестів. Поєднання pytest з методами обробки природної мови дозволяє формалізувати перевірку якісних характеристик текстового контенту, таких як граматична коректність, стилістична узгодженість та відповідність заданим правилам.

Важливо розрізняти тестування “класичних” алгоритмів і тестування компонентів, що використовують підходи обробки природної мови або елементи ШІ. У першому випадку очікувані результати часто детерміновані, тоді як у задачах NLP можуть бути допустимі множини правильних відповідей або ситуації, де результат оцінюється якісно. У таких системах ефективність тестування забезпечується не спробою звести складну поведінку до одного

“правильного значення”, а конструюванням критеріїв прийнятності: порогів, профілів помилок, вагових коефіцієнтів, пріоритизації дефектів за критичністю. Для прикладних задач контролю якості тексту це природно реалізується через класифікацію мовних помилок, де критичні помилки формують підставу для падіння тесту, а некритичні — для попереджень і накопичуваного звіту (що узгоджується з практичною частиною роботи)[37].

Цей підхід принципово відрізняється від “банального” тестування, оскільки переводить автоматизацію з рівня перевірки інтерфейсів на рівень перевірки якості даних. У межах обраної теми саме така постановка задачі дає можливість оцінити ефективність автоматизованого тестування: по-перше, за стабільністю результатів (повторюваність перевірок на тому самому корпусі текстів), по-друге, за корисністю сигналу (відношення критичних знахідок до шуму), і, по-третє, за економією часу редакційного процесу, коли автоматизовані перевірки виконуються швидше за ручну коректуру на базовому рівні.

Таким чином, перспективи використання pytest виходять за межі традиційного вебтестування. Фреймворк може розглядатися як універсальний інструмент автоматизованого тестування нефронтендних програмних систем, що поєднує класичні підходи до тестування з сучасними методами аналізу даних та штучного інтелекту.

Обґрунтування використання PyTest для тестування нефронтендних та інтелектуальних систем.

Аналіз існуючих підходів до автоматизованого тестування свідчить про доцільність використання універсальних фреймворків, здатних адаптуватися до різних типів програмних систем. У цьому контексті PyTest може розглядатися як ефективна основа для побудови систем автоматизованого тестування нефронтендних та інтелектуальних програмних рішень.

Оскільки тема роботи включає не лише розробку підходу, але й оцінку його ефективності, необхідно визначити, що саме розуміється під “ефективністю” автоматизованого тестування. У практично орієнтованих дослідженнях ефективність доцільно трактувати як поєднання результативності (здатності

виявляти значущі дефекти), економічності (зменшення витрат часу та ресурсів у порівнянні з ручними перевірками) і надійності (стабільність результатів прогонів і низька частка хибних спрацювань). Такий підхід є узгодженим з ідеєю контролю якості як процесу, де важливо не просто “мати тести”, а отримувати швидкий, точний і діагностичний зворотний зв’язок[38].

У межах даної роботи оцінка ефективності автоматизованого тестування на базі pytest логічно спирається на інструменти вимірювання, що доповнюють сам фреймворк. З одного боку, показники покриття коду (coverage) відображають ширину перевірок, а з іншого — звітність (наприклад, HTML-репорти) відображає якість діагностики та придатність тестових результатів для практичного використання. Для задач аналізу тексту ефективність також включає контроль “шуму” в результатах: тестова політика повинна враховувати, що NLP-інструменти можуть формувати некритичні зауваження, які не є дефектами з точки зору редакційного процесу, тому критерії успішності повинні бути пороговими та прив’язаними до критичності помилок[39].

Гнучка архітектура PyTest дозволяє реалізовувати тестові сценарії, орієнтовані на перевірку логіки програмних компонентів незалежно від наявності графічного інтерфейсу. Завдяки системі фікстур і підтримці плагінів PyTest може інтегруватися з інструментами аналізу природної мови та іншими компонентами штучного інтелекту, що дає змогу формалізувати перевірку якісних характеристик результатів.

Таким чином, використання PyTest у межах даної роботи дозволяє не лише розробити систему автоматизованого тестування, а й оцінити її ефективність у контексті сучасних вимог до якості програмного забезпечення. Застосування такого підходу створює передумови для розширення автоматизованого тестування за межі традиційних вебзастосунків і підтверджує актуальність обраної теми дослідження.

РОЗДІЛ 2. АНАЛІЗ ТА ПРОЄКТУВАННЯ ПІДХОДУ ДО АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ДЕСКТОПНИХ ТА ІНТЕЛЕКТУАЛЬНИХ ПРОГРАМНИХ СИСТЕМ

2.1. Аналіз предметної області та існуючих підходів до тестування нефронтендних систем

Нефронтендні програмні системи становлять значну частину сучасного програмного забезпечення та охоплюють широкий спектр прикладних рішень, починаючи від десктопних застосунків і фонових сервісів до аналітичних систем і модулів обробки даних. На відміну від веборієнтованих застосунків, такі системи часто не мають вираженого користувацького інтерфейсу або характеризуються обмеженою взаємодією з користувачем, що суттєво впливає на підходи до їх тестування[29].

У наукових джерелах зазначається, що класичні методи автоматизації тестування, орієнтовані на перевірку графічного інтерфейсу, є недостатньо ефективними для нефронтендних систем. UI-орієнтовані тести мають високу залежність від змін візуальних компонентів, що призводить до зростання витрат на їх підтримку та зниження стабільності результатів тестування[30].

Альтернативним підходом, який набуває все більшого поширення, є тестування бізнес-логіки та алгоритмічних компонентів програмних систем. Такий підхід передбачає фокусування на перевірці коректності обробки даних, виконання правил предметної області та стабільності функціонування внутрішніх модулів незалежно від способу взаємодії з користувачем[31].

Для нефронтендних систем характерним є змішаний профіль тестування, у якому різні рівні перевірок дають різну цінність для контролю якості. На рівні графічного інтерфейсу тести можуть відтворювати дії користувача, однак вони є найдорожчими з точки зору супроводу і найменш стабільними при зміні середовища. Для десктопних застосунків у Windows до типових причин нестабільності належать залежність від фокусу вікна, роздільної здатності, продуктивності робочої станції, фонового навантаження та різниці конфігурацій

ОС. Унаслідок цього UI-автоматизація доцільна як інструмент димового контролю ключових сценаріїв, але не як основа вимірювання ефективності тестування логіки системи.

У практиці тестування десктопних застосунків у Windows поширені підходи, що базуються на механізмах Windows UI Automation або сумісних протоколах взаємодії з інтерфейсом. Вони дозволяють програмно знаходити елементи керування, ініціювати дії та виконувати елементарні перевірки стану інтерфейсу. Проте навіть за наявності таких механізмів загальна вартість супроводу UI-набору зростає швидше, ніж у випадку модульних або інтеграційних тестів. Звідси випливає інженерно обґрунтована стратегія: зменшувати частку UI-тестів на користь тестів, що перевіряють внутрішню логіку і компоненти обробки даних, залишаючи UI-рівень як верхівку набору тестів.

Окремого значення набувають підходи, у яких нефронтендна система тестується через “стабільні інтерфейси” — файлові формати, API-виклики, міжпроцесну взаємодію, журнали подій або контрольовані точки інтеграції.. Для задач даної роботи це принципово, оскільки об’єктом перевірки виступає не UI як такий, а коректність обробки текстового контенту і стабільність логіки перевірок у середовищі Windows[40].

Практичний висновок з аналізу існуючих підходів полягає в тому, що тестова стратегія для нефронтендних систем повинна бути багаторівневою: модульні тести забезпечують швидкий і стабільний сигнал про дефекти, інтеграційні тести перевіряють коректність взаємодії компонентів (наприклад, витяг тексту з PDF та передача його в модуль аналізу), тоді як UI-тести виконують роль обмеженого регресійного контролю основних користувацьких потоків. Саме така конфігурація дозволяє досягти відтворюваності й керованості тестування, а також створює передумови для оцінювання ефективності автоматизації як процесу, а не як сукупності розрізнених тестів(див.табл.2.1.)[29].

Таблиця 2.1

Порівняння поширених підходів та інструментів автоматизації тестування десктопних систем у Windows

Підхід / інструмент	**Технічна основа**	**Сильні сторони**	**Обмеження**	**Доцільність у межах цієї роботи**
UI Automation (Windows)	API доступу до UI-елементів ОС	Реалістичні сценарії користувача, smoke-контроль	Нестабільність, висока вартість підтримки	Обмежено, лише для ключових smoke-перевірок
WinAppDriver / WebDriver-подібні	Автоматизація UI через драйвер	Уніфікований стиль E2E, кросмовність	Залежність від середовища, складність налаштування	Додатково, якщо потрібен UI-рівень
pywinauto-подібні бібліотеки	Win32/UIA автоматизація	Швидкі прототипи для desktop	Залежність від стану ОС, таймінги	Додатково, для прикладних демонстрацій
Компонентні тести (pytest)	Виклик коду без UI	Висока відтворюваність, швидкість, діагностичність	Потрібна архітектурна придатність коду	Основний підхід, ядро автоматизації
Інтеграційні тести через файли/дані	“Стабільні інтерфейси”	Менше flaky-ризиків, ближче до	Потрібен контроль тестових даних	Основний підхід для тексту/PDF

Підхід / інструмент	Технічна основа	Сильні сторони	Обмеження	Доцільність у межах цієї роботи
		реального процесу		

Джерело: [авторська таблиця]

Особливої уваги потребують програмні системи, що працюють з неструктурованими або слабкоструктурованими даними, зокрема текстовими матеріалами. У таких випадках тестування виходить за межі перевірки детермінованих результатів і потребує використання метрик якості, що враховують ймовірнісний характер обробки інформації.

У межах даного дослідження доцільно узагальнити існуючі підходи до тестування нефронтендних систем у вигляді концептуальної схеми (див. рис.2.1.).

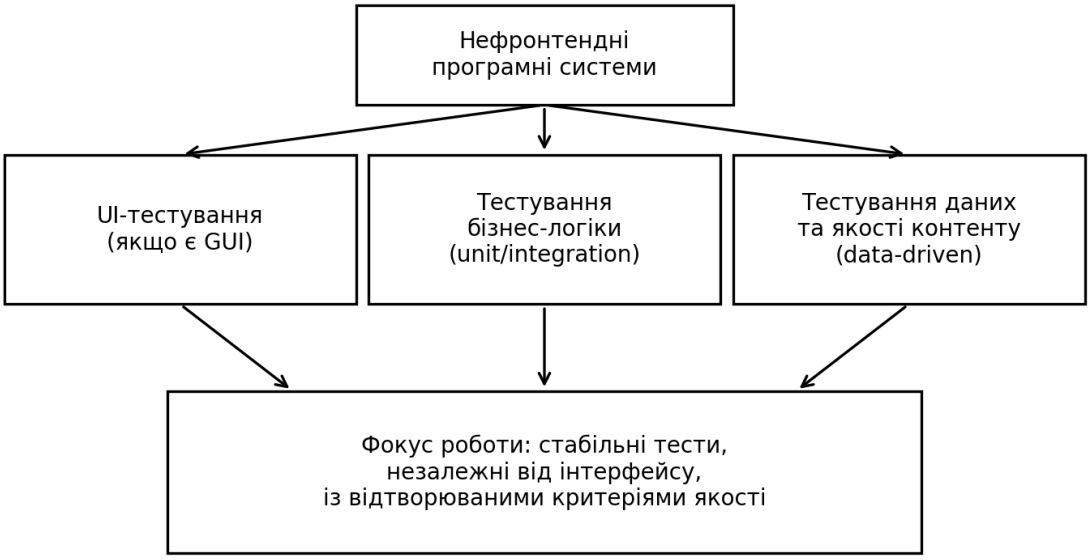


Рис. 2.1. Узагальнені підходи до тестування нефронтендних програмних систем

Джерело: [авторський рисунок]

2.2. Проєктування моделі автоматизованого тестування на базі pytest

Проєктування моделі автоматизованого тестування для нефронтендних програмних систем передбачає визначення архітектурних компонентів, потоків даних та механізмів оцінювання результатів тестування. У межах даної роботи pytest розглядається як центральний елемент тестової інфраструктури, який забезпечує запуск тестів, керування залежностями та формування результатів перевірки[27].

Запропонована модель ґрунтується на модульному підході, відповідно до якого тестові сценарії структуруються за функціональними ознаками та ізольовано перевіряють окремі аспекти поведінки системи. Така організація дозволяє зменшити зв’язаність між тестами та підвищити їхню стійкість до змін у програмному коді.

Ключову роль у моделі відіграють фікстури pytest, які використовуються для ініціалізації тестового середовища, підготовки вхідних даних та керування зовнішніми залежностями. Застосування фікстур забезпечує повторюваність тестових сценаріїв і відповідає рекомендаціям щодо побудови автоматизованих тестів, викладеним у працях з шаблонів тестування[25].

Для того щоб модель автоматизованого тестування на базі pytest була придатною до оцінювання ефективності, вона має визначати не лише спосіб запуску тестів, а й структуру артефактів, які утворюються внаслідок тестових прогонів, а також правила інтерпретації результатів. У межах даної роботи система тестування розглядається як конвеєр, де вхідні дані (текстові файли та PDF-документи) проходять етапи підготовки, аналізу та прийняття тестового рішення. У такій постановці pytest виконує роль керуючого механізму, який забезпечує відтворюваність прогонів, ізоляцію тестів, повторне використання компонентів через фікстури та формалізований висновок про успішність перевірки[26].

Ключовим компонентом проєктної моделі є керування тестовими даними. Для задач контролю якості тексту тестовий набір повинен містити як приклади “нормативних” матеріалів (очікуваний мінімальний рівень помилок), так і

приклади з навмисно внесеними дефектами (орфографічні, граматичні, пунктуаційні, стилістичні). Такий підхід дозволяє перевіряти не лише відсутність критичних помилок у “позитивних” зразках, а й чутливість системи до типових дефектів. Практично це забезпечує можливість оцінювати ефективність автоматизації не на рівні декларацій, а на рівні вимірюваних результатів: які типи помилок виявляються стабільно, які формують шум, і як це впливає на рішення тестів.

Важливою складовою моделі є модуль звітності. Результати автоматизованого тестування мають бути не лише придатними для інтерпретації розробником, а й зрозумілими як артефакт контролю якості, який може бути використаний у процесах безперервної інтеграції. У цьому контексті ефективність тестування залежить від діагностичності: тест повинен не просто “падати”, а фіксувати тип проблеми, контекст, джерело помилки та її критичність. Це особливо актуально для задач аналізу тексту, де одна й та сама помилка може проявлятися у різних формах, а частина повідомлень інструментів аналізу може бути некритичною для виробничого процесу[27].

Оскільки оцінювання ефективності тестування є частиною теми роботи, модель повинна передбачати метрики та спосіб їх отримання. Доцільно спиратися на підхід Goal–Question–Metric, у межах якого формулюється ціль (наприклад, скорочення ручної перевірки), уточнюються питання (яка частка дефектів виявляється автоматично, який відсоток шуму) та визначаються метрики (час прогону, частота хибних спрацювань, частка критичних помилок)[28]. Така методологія дозволяє пов’язати технічні параметри тестів з практичною корисністю рішення (див.табл.2.2.)

Таблиця 2.2

Відповідність компонентів моделі автоматизованого тестування та засобів реалізації

Компонент моделі	**Призначення**	**Реалізація в підході роботи**	**Артефакт/результат**
Оркестрація тестів	Запуск і керування сценаріями	pytest (запуск, маркери, параметризація) [7]	Статус тестів, лог прогону
Підготовка середовища	Ізоляція, повторюваність	fixtures pytest [7; 17]	Стандартизований контекст тесту
Тестові дані	Репрезентативність перевірок	Корпус текстів/PDF, позитивні/негативні зразки	Відтворювана база для регресії
Аналіз тексту	Виявлення мовних дефектів	LanguageTool/spaCy як модулі аналізу [13; 12]	Перелік помилок/ознаки
Інтерпретація	Прийняття рішення тестом	Пороги, критичність, агрегування	Pass/Fail + пояснення
Звітність	Діагностика і контроль	Форматований звіт для перегляду	Зрозумілий результат тестування

Джерело: [авторська таблиця]

Окремим елементом моделі є модулі обробки результатів тестування, які дозволяють інтерпретувати результати перевірок з урахуванням специфіки нефронтендних систем. У випадку тестування алгоритмів аналізу тексту результати можуть мати нечіткий або ймовірнісний характер, що потребує використання порогових значень та агрегованих метрик.

Для наочного представлення запропонованої моделі доцільно використати архітектурну схему, яка відображає взаємодію між тестовими сценаріями,

фікстурами pytest та зовнішніми інструментами аналізу даних (див.рис.2.2. та табл.2.3.).

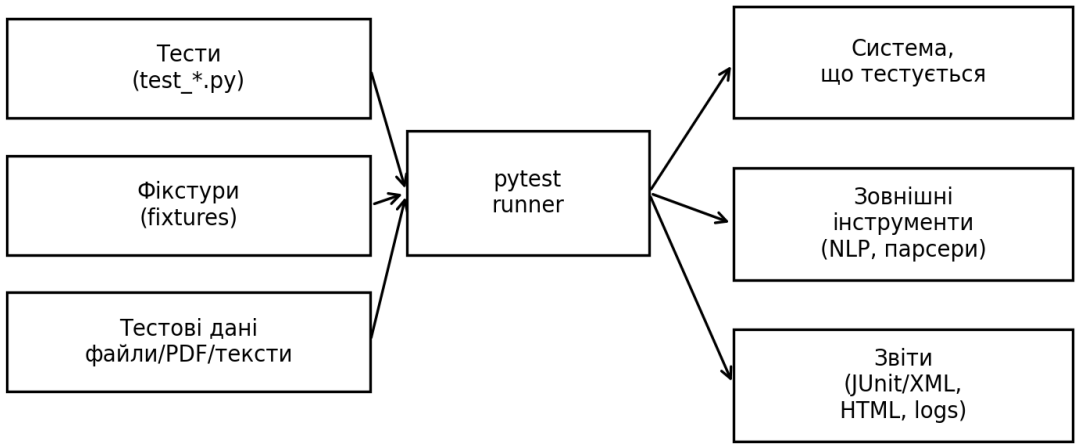


Рис. 2.2 Архітектура моделі автоматизованого тестування на базі pytest.

Джерело: [авторський рисунок]

Таблиця 2.3

Метрики для оцінювання ефективності автоматизованого тестування на базі pytest

Метрика	**Суть метрики**	**Як вимірюється в межах роботи**	**Інтерпретація для висновків**
Час прогону тестів	Швидкість отримання зворотного зв’язку	Середній/медіанний час прогону набору	Менше — краще для CI і регресії
Стабільність прогонів	Частка “flaky” падінь	Частота нестабільних тестів на однакових даних	Менше — вища довіра до тестів
Діагностичність	Наскільки легко знайти причину падіння	Наявність типу помилки, контексту, критичності	Більше — швидше усунення дефектів

Метрика	Суть метрики	Як вимірюється в межах роботи	Інтерпретація для висновків
Частка критичних помилок	“Сигнал” проти “шуму”	Кількість критичних помилок / усі повідомлення	Вища частка = точніші критерії
Покриття перевірок даних	Ширина контролю контенту	Частка правил/критеріїв, що застосовуються	Більше — ширший контроль якості
Економія ручної перевірки	Практичний ефект	Орієнтовна різниця часу “до/після”	Підтверджує практичну значущість

Джерело: [авторська таблиця]

Інтеграція методів обробки природної мови та елементів штучного інтелекту.

Інтеграція методів обробки природної мови у процес автоматизованого тестування є обґрунтованою відповіддю на зростання кількості програмних систем, що працюють з неструктурованими текстовими даними. На відміну від класичних програмних компонентів, результати роботи таких систем часто мають імовірнісний характер, що ускладнює застосування детермінованих критеріїв успішності тестів[12].

У наукових дослідженнях з обробки природної мови зазначається, що якість текстового контенту може оцінюватися за сукупністю показників, зокрема граматичною коректністю, стилістичною узгодженістю та відповідністю мовним нормам. Формалізація таких показників створює можливість їх використання в автоматизованих тестових сценаріях[9].

У межах даної роботи pytest використовується як інструмент оркестрації тестових сценаріїв, у яких результати аналізу природної мови, отримані за допомогою спеціалізованих бібліотек та сервісів, інтегруються у процес

прийняття рішення щодо успішності тесту. Такий підхід дозволяє поєднати класичні методи тестування з інтелектуальними алгоритмами аналізу тексту.

Застосування інструментів, таких як LanguageTool та бібліотеки spaCy, дозволяє реалізувати перевірку текстових матеріалів на наявність граматичних і стилістичних помилок, а також виконувати попередню лінгвістичну обробку даних. Отримані результати можуть бути агреговані та інтерпретовані з використанням порогових значень, що забезпечує адаптацію тестових сценаріїв до специфіки імовірнісних моделей[11].

Інтеграція NLP-інструментів у тестові сценарії потребує коректного трактування результатів аналізу. На відміну від класичних тестів, де очікуваний результат часто є однозначним, мовний аналіз працює з багатьма гіпотезами і може формувати зауваження різної природи. Частина з них відображає реальні помилки, частина — стилістичні рекомендації або спрацювання, чутливі до контексту. Тому для забезпечення відтворюваності й корисності тестів необхідно встановити політику інтерпретації, яка відділяє критичні помилки від некритичних повідомлень. У межах даного підходу тестове рішення формується не “за фактом будь-якого зауваження”, а за пороговою логікою з урахуванням типу і критичності помилки, що дозволяє мінімізувати хибнопозитивні спрацювання та підвищити довіру до тестового сигналу.

Окремо слід враховувати вплив версійності інструментів аналізу. Оновлення правил, моделей або словників можуть змінювати характер виявлення помилок навіть за незмінного тестового набору. Для задачі оцінювання ефективності автоматизованого тестування це означає необхідність фіксації версій середовища (інтерпретатор, бібліотеки, правила перевірок), інакше порівняння результатів у часі втрачає коректність. За таких умов pytest виступає не лише “раннером тестів”, а й механізмом стандартизації прогонів, що робить можливим порівняння метрик ефективності між різними ітераціями рішення.

Для наочного представлення взаємодії між фреймворком pytest та інструментами обробки природної мови доцільно використати концептуальну схему інтеграції (див.рис.2.3.).

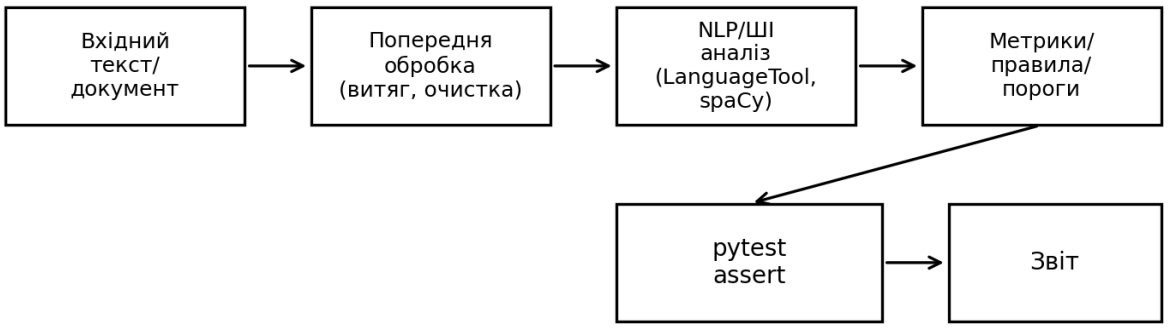


Рисунок 2.3. Схема інтеграції pytest
з методами обробки природної мови та ШІ

Джерело: [авторський рисунок]

Отже, використання методів обробки природної мови у поєднанні з фреймворком pytest дозволяє розширити клас задач автоматизованого тестування та забезпечити контроль якості програмних систем, орієнтованих на аналіз текстових даних.

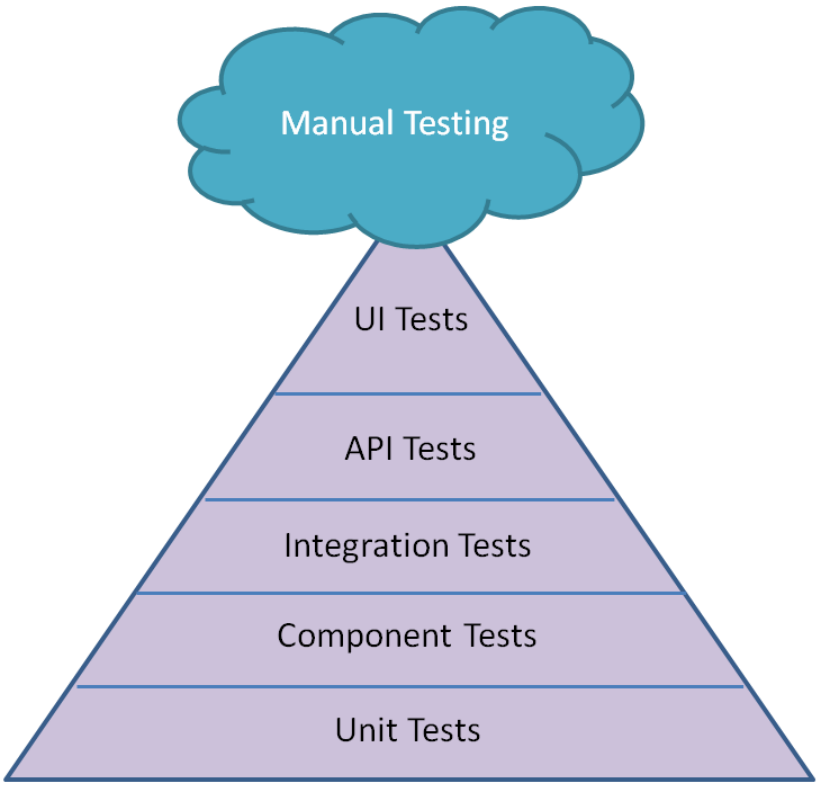


Рис. 2.4. Узагальнена «піраміда тестування» для нефронтендної системи контролю якості тексту

Джерело: [онлайн видавництво Medium]

РОЗДІЛ 3 ПРАКТИЧНА РЕАЛІЗАЦІЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НЕФРОНТЕНДНИХ ПРОГРАМНИХ СИСТЕМ НА БАЗІ PYTEST

3.1. Характеристика об’єкта та середовища практичної реалізації

Практична частина кваліфікаційної роботи ґрунтується на використанні реального виробничого середовища, у межах якого здійснюється підготовка, обробка та перевірка текстового контенту. Об’єктом практичної реалізації є програмна система автоматизованого контролю якості текстових матеріалів, що функціонує у середовищі операційної системи Windows та орієнтована на підтримку внутрішніх редакційних процесів.

Особливістю даного середовища є поєднання десктопних програмних засобів, локальних обчислювальних ресурсів і скриптових компонентів, реалізованих мовою Python. Основні операції з обробки текстів виконуються без використання вебінтерфейсу, що зумовлює доцільність застосування нефронтендного підходу до автоматизованого тестування.

У межах практичної реалізації використовувався програмний стек, який включає інтерпретатор Python, фреймворк pytest, бібліотеки для роботи з текстовими та PDF-документами, а також інструменти аналізу природної мови. Така конфігурація середовища дозволяє реалізувати автоматизовані перевірки як функціональних, так і якісних характеристик текстового контенту.

Для забезпечення відтворюваності практичної реалізації середовище виконання було стандартизовано шляхом ізоляції залежностей і фіксації конфігурації запуску. Інструментальна частина рішення реалізована мовою Python та запускається у вигляді окремого тестового проєкту, де pytest виконує роль тестового раннера й оркестратора сценаріїв перевірки. Така організація дозволяє відокремити логіку аналізу тексту (як об’єкт тестування) від механізму запуску тестів (як засобу контролю), а також забезпечити повторюваність прогонів у межах одного робочого середовища Windows.

Окремо було враховано специфіку задачі контролю якості текстового контенту, що включає роботу з різними форматами вхідних матеріалів. У межах практичної реалізації джерелами вхідних даних виступають текстові файли та PDF-документи, які проходять етапи витягу тексту, нормалізації та лінгвістичного аналізу. Таким чином, тестова система фактично працює як конвеєр, у якому pytest запускає відтворювані перевірки для набору документів і формує результат у вигляді статусів тестів та звітів, придатних для подальшого аналізу ефективності (див.табл.3.1.).

Таблиця 3.1

Конфігурація середовища практичної реалізації та програмного стеку

Компонент	**Значення (вписати фактичне)**	**Призначення**
Операційна система	Windows 10/11 (версія)	Виконання десктопних сценаріїв і запуск тестів
Python	3.x	Середовище виконання тестів та утиліт
pytest	x.x	Оркестрація та виконання тестових сценаріїв
LanguageTool	локально/через API (версія)	Правило-орієнтована перевірка граматики та орфографії
spaCy + модель	uk_core_news_sm (версія)	Структурний NLP-аналіз тексту
pytest-cov + coverage.py	версії	Оцінювання покриття та звітність
pytest-html	версія	Формування HTML-звітів тестового прогону

Джерело: [авторська таблиця]

Для наочного відображення структури об’єкта практичної реалізації доцільно використати узагальнену схему середовища.

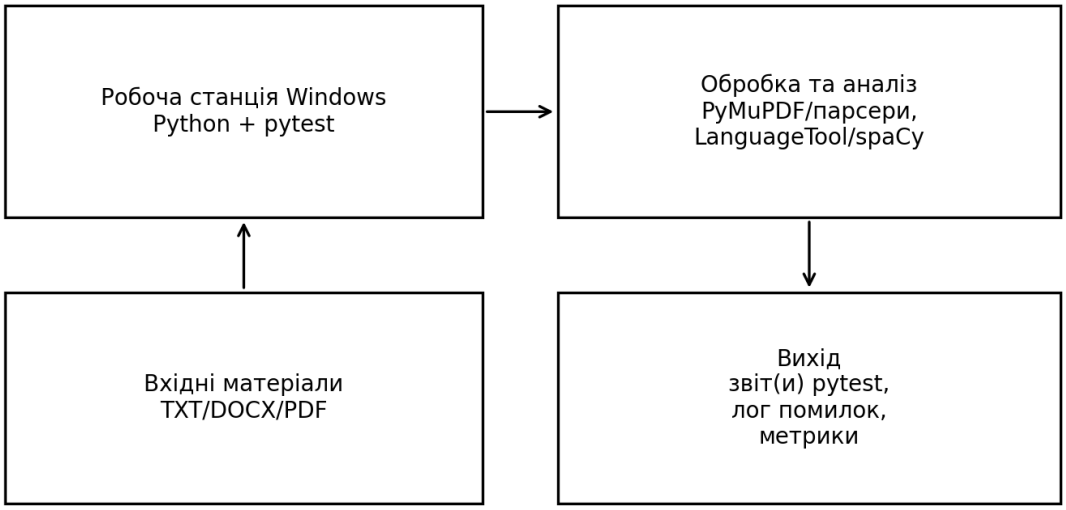


Рис. 3.1 Загальна схема об’єкта та середовища практичної реалізації.

Джерело: [авторський рисунок]

3.2. Проєктування та реалізація автоматизованих тестових сценаріїв

Проєктування автоматизованих тестових сценаріїв у межах практичної реалізації ґрунтується на принципі ізоляції логіки перевірки від конкретних засобів взаємодії з користувачем. Такий підхід дозволяє застосовувати фреймворк pytest для тестування нефронтендних програмних систем незалежно від наявності або типу інтерфейсу.

Автоматизовані тестові сценарії були організовані у вигляді модульної структури, де кожен тест відповідає за перевірку окремого аспекту якості текстового контенту. До таких аспектів належать коректність обробки вхідних даних, відповідність встановленим правилам оформлення та відсутність критичних мовних помилок.

Фреймворк pytest використовувався як центральний механізм запуску тестів, керування тестовими залежностями та формування звітів про результати перевірок. Застосування фікстур дозволило уніфікувати процес підготовки

тестових даних та забезпечити відтворюваність виконання тестових сценаріїв у різних середовищах.

Практична реалізація тестових сценаріїв була організована таким чином, щоб кожна перевірка відображала окремий етап конвеєра обробки документів і давала діагностичний результат у разі відхилення від критеріїв якості. У межах підходу було зафіксовано принцип: тест повинен завершуватися з помилкою лише тоді, коли відхилення має практично значущу критичність. Для задач лінгвістичного контролю це означає, що орфографічні та граматичні помилки розглядаються як критичні, а пунктуаційні чи стилістичні зауваження можуть трактуватися як попередження або інформаційні сигнали, що не блокують процес.

Для забезпечення масштабованості тестовий набір був параметризований за вхідними документами. Один і той самий тестовий сценарій застосовується до набору файлів, що репрезентує різні типи матеріалів, а підготовка доступу до даних реалізована через фікстури pytest. Це дозволяє збільшувати корпус документів без зміни логіки тестів, зберігаючи при цьому відтворюваність прогонів і простоту супроводу. Такий підхід узгоджується з можливостями pytest щодо параметризації та керування життєвим циклом ресурсів через фікстури (див.табл3.2.)[27].

Таблиця 3.2

Логічна структура тестового набору та призначення сценаріїв

Група тестів	**Об’єкт перевірки**	**Вхідні дані**	**Результат перевірки**
Тести підготовки даних	Доступність файлів, кодування, базова валідація	TXT/PDF	Помилка при некоректних вхідних даних
Тести витягу тексту	Коректність отримання тексту з документів	PDF	Вихідний текст придатний до аналізу

Група тестів	Об’єкт перевірки	Вхідні дані	Результат перевірки
Тести LanguageTool	Граматика/орфографія, правило-орієнтовані перевірки	Текст	Pass/Fail за порогом критичних помилок
Тести spaCy	Структурні властивості тексту (речення, токени)	Текст	Pass/Fail за структурними критеріями
Тести регресії	Порівняння стабільності на фіксованому корпусі	Корпус	Відсутність деградації результатів

Джерело: [авторська таблиця]

У процесі реалізації тестових сценаріїв здійснювалася автоматизована обробка текстових файлів і PDF-документів. Отриманий текстовий контент передавався до модулів аналізу, де перевірявся відповідно до визначених критеріїв якості.

Для наочного представлення структури автоматизованих тестових сценаріїв доцільно використати узагальнену схему ((див.рис.3.2.).

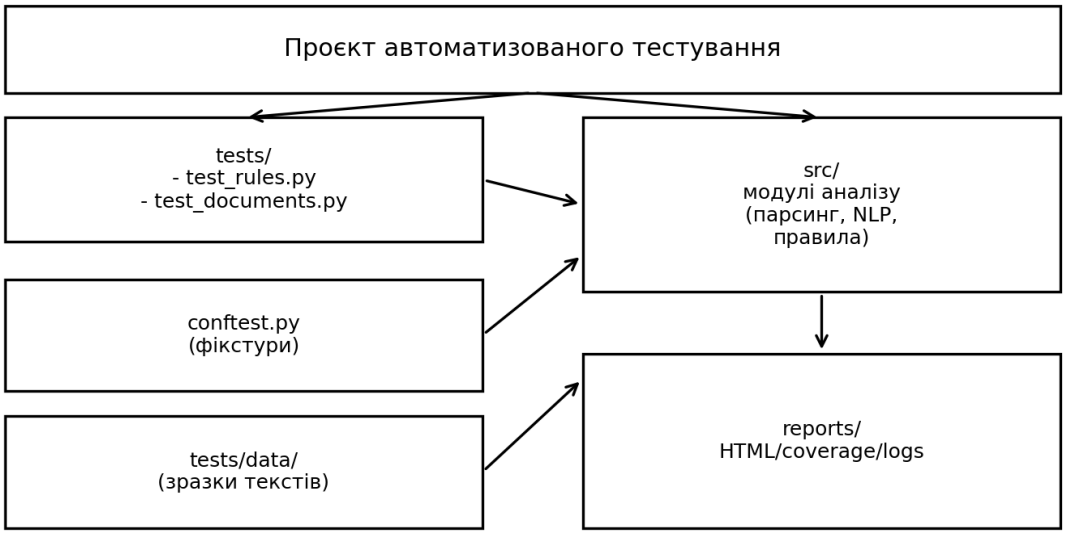


Рис. 3.2 Структура автоматизованих тестових сценаріїв на базі pytest

Джерело: [авторський рисунок]

Таким чином, реалізований підхід до проєктування тестових сценаріїв забезпечує масштабованість системи автоматизованого тестування та створює передумови для її подальшого розширення.

Алгоритм виконання тестового прогону в межах розробленого підходу має послідовну структуру. На першому етапі формується набір вхідних документів і здійснюється їх нормалізація до уніфікованого текстового подання. Далі запускається лінгвістичний аналіз: правило-орієнтована перевірка LanguageTool формує перелік знайдених відхилень з типами правил і позиційними характеристиками, а spaCy використовується для отримання структурних ознак тексту, які можуть виступати контрольними критеріями якості. На наступному етапі результати аналізу інтерпретуються: за кожним документом обчислюються агреговані показники (кількість критичних помилок, кількість некритичних зауважень), після чого формується тестове рішення за визначеною пороговою політикою. Пороговий підхід забезпечує придатність результатів для регресійного контролю, оскільки дозволяє стабільно відділяти критичні дефекти від “шумових” повідомлень, що не є блокуючими у виробничому процесі.

Для формування звітності тестовий прогін може доповнюватися інструментами репортингу та оцінки покриття. Використання pytest-html дозволяє отримати єдиний артефакт прогону у вигляді HTML-звіту, а зв’язка pytest-cov і coverage.py дає можливість формувати звіти про покриття коду та оцінювати ширину тестових перевірок у термінах інструментальної метрики[23]. Команди запуску та приклади конфігурації доцільно винести в додатки у вигляді листингів, щоб забезпечити однозначне відтворення експерименту.

Для демонстрації того, що pytest може застосовуватись не лише для вебперевірок, у практичній частині передбачено приклад димового тесту десктопного сценарію у Windows. Такий тест не підміняє модульні перевірки логіки і не масштабується як основа регресії, однак підтверджує можливість запуску UI-автоматизації у межах єдиного pytest-проєкту, де результати UI-рівня потрапляють до загального звіту тестового прогону. Реалізаційно подібні сценарії будуються поверх механізмів UI Automation і можуть виконуватися як

окрема позначена група тестів, наприклад з маркером, щоб не впливати на швидкість базового регресійного набору.

Інтеграція інструментів обробки природної мови.

Інтеграція методів обробки природної мови є ключовим елементом практичної реалізації, оскільки дозволяє перейти від формальних перевірок до оцінювання якісних характеристик текстового контенту. У межах рішення фреймворк pytest виконує роль оркестратора перевірок, а лінгвістичний аналіз здійснюється спеціалізованими інструментами, результати яких інтерпретуються як критерії успішності тестів [7].

Для автоматизованого виявлення орфографічних, граматичних і частини стилістичних помилок використовується LanguageTool, який надає правила та механізми перевірки природномовних текстів і може застосовуватися як локально, так і через API. Важливою перевагою такого підходу є відтворюваність: однаковий вхідний текст за фіксованих налаштувань повертає зіставні результати, що придатно для регресійного контролю.

Результат перевірки LanguageTool технічно подається у вигляді структури, що містить перелік знайдених збігів правил з описом типу проблеми, позицією в тексті, повідомленням та можливими варіантами виправлення. Для включення такого результату у тестовий сценарій pytest необхідно виконати два кроки: по-перше, трансформувати “сирі” збіги у класифіковані події тестування, по-друге, застосувати порогову політику прийняття рішення. Класифікація реалізується через віднесення знайдених збігів до груп критичності на підставі їх типу (наприклад, граматика й орфографія як критичні, пунктуація як середня критичність), після чого для документа обчислюється агрегована кількість критичних подій. У випадку перевищення порогу тест завершується невдачею, а діагностика включає тип помилки та короткий контекст, що підвищує практичну придатність результатів(див.табл.3.3.).

Таблиця 3.3

Правила інтерпретації результатів LanguageTool у тестових сценаріях

Категорія результату	**Типи (приклад)**	**Рівень критичності**	**Вплив на тест**
Орфографія	misspelling	Високий	Fail при наявності або при перевищенні порогу
Граматика	grammar	Високий	Fail при наявності або при перевищенні порогу
Пунктуація	punctuation	Середній	Логування/звіт, не блокує при малих значеннях
Стиль	style	Середній/низький	Логування/звіт
Інше	typographical, uncategorized	Низький	Інформаційне повідомлення

Джерело: [авторська таблиця]

Для попередньої лінгвістичної обробки даних і додаткових перевірок застосовуються NLP-бібліотеки (зокрема spaCy), які забезпечують токенізацію, лематизацію та морфологічний аналіз. Це дозволяє будувати перевірки на рівні структурних властивостей тексту (наприклад, наявність обов’язкових елементів, узгодженість форм, контроль частотності певних конструкцій) і доповнювати правило-орієнтовані перевірки.

На відміну від правило-орієнтованих перевірок, spaCy використовується для отримання структурних характеристик тексту, які можуть бути формалізовані у вигляді чітких критеріїв. Практично це включає перевірки мінімальної структурної повноти матеріалу, наприклад, наявність достатньої кількості речень або контроль базових шаблонів оформлення, що зменшує ризик пропуску “технічних” дефектів контенту, не пов’язаних безпосередньо з орфографією. Важливим моментом є фіксація моделі та її версії, оскільки зміна

моделі може змінювати результати токенізації та сегментації, що безпосередньо впливає на відтворюваність регресійного контролю[26].

Оскільки результати лінгвістичного аналізу можуть містити хибнопозитивні спрацювання, у тестових сценаріях застосовується порогова інтерпретація: помилки класифікуються за критичністю, а успішність тесту визначається не лише фактом наявності помилок, а й їх кількістю та типом. Такий підхід дозволяє формалізувати “якісні” критерії і зробити їх придатними для автоматизованої перевірки у pytest[27].

Для забезпечення повторного використання компонентів аналізу застосовуються фікстури, які ініціалізують конфігурацію інструментів NLP, завантажують правила перевірок та уніфікують отримання результатів. Це спрощує супровід тестів і підвищує їхню масштабованість при зростанні кількості перевірюваних матеріалів (див.рис.3.3. та табл.3.4.).

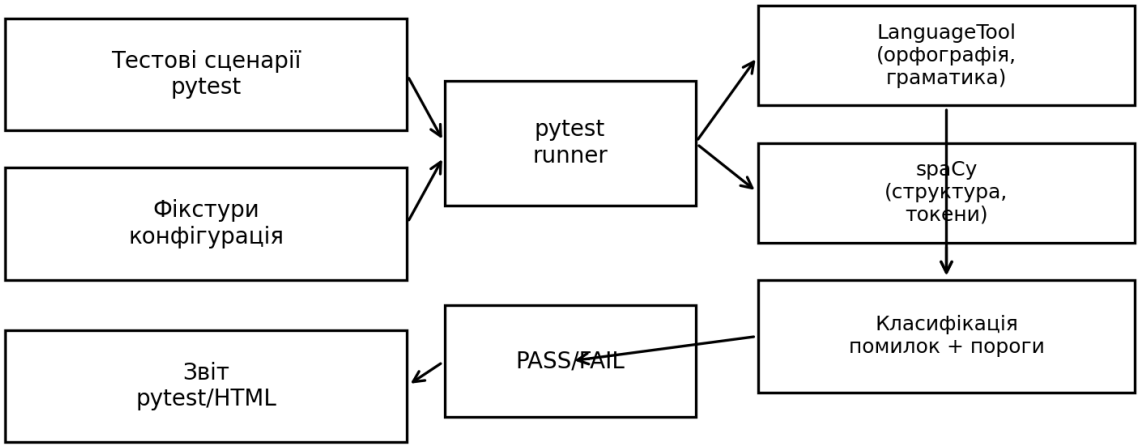


Рис. 3.3 Схема інтеграції pytest з інструментами обробки природної мови.

Джерело: [авторський рисунок]

Таблиця 3.4

Приклад класифікації мовних помилок за критичністю

Тип мовної помилки	**Опис помилки**	**Приклад прояву**	**Рівень критичності**	**Реакція автоматизованого тесту**
Орфографічна	Неправильне написання слова, яке порушує мовні норми	*інформациія, програмне забеспечення*	Високий	Тест не пройдено (fail)
Граматична	Помилка у відмінюванні, узгодженні слів або побудові речення	*дані було опрацьовані неправильно*	Високий	Тест не пройдено (fail)
Пунктуаційна	Неправильне використання розділових знаків	Відсутня кома у складному реченні	Середній	Попередження (warning)
Стилістична	Порушення стилістичних норм або невідповідність жанру	Розмовні слова в інформаційній статті	Середній	Попередження (warning)

Джерело: [авторська таблиця]

Аналіз результатів та практична значущість.

Оцінювання ефективності практичної реалізації виконувалось як серія контрольованих прогонів тестового набору на фіксованому корпусі документів. Методика оцінювання спирається на поєднання інженерних і процесних показників: тривалість тестового прогону як характеристика швидкості зворотного зв’язку; стабільність результатів як ознака відсутності “flaky”-поведінки; діагностичність як здатність тестів пояснювати причину невдачі;

частка критичних помилок серед усіх знайдених повідомлень як показник співвідношення сигналу до шуму. Окремо оцінюється практичний ефект у вигляді скорочення ручної перевірки за рахунок автоматичного відсіву матеріалів з критичними дефектами (див.табл.3.5.).

Таблиця 3.5

Зведені результати контрольних прогонів та показники ефективності

Показник	Значення	Коментар
Кількість документів у корпусі		TXT+PDF
Кількість тестів у наборі		Модульні + інтеграційні
Середній час прогону		Окремо локально / CI за потреби
Частка нестабільних падінь		Flaky rate
Середня кількість критичних помилок на документ		За політикою критичності
Частка критичних повідомлень серед усіх		Signal/Noise
Оцінка економії ручної перевірки		До/після автоматизації

Джерело: [авторська таблиця]

Аналіз результатів практичної реалізації свідчить про ефективність використання pytest для автоматизованого тестування нефронтендних програмних систем. Запропонований підхід дозволяє зменшити обсяг ручної перевірки текстових матеріалів, підвищити стабільність процесів контролю якості та забезпечити відтворюваність результатів тестування.

Практична значущість отриманих результатів полягає у можливості використання системи автоматизованого тестування у реальних виробничих умовах, а також у можливості адаптації підходу до інших типів нефронтендних програмних систем і задач аналізу даних.

Окремою перевагою рішення є можливість вбудовування тестів у процеси безперервної інтеграції: тести можуть виконуватися за розкладом або при появі нових матеріалів, а результати зберігатися у форматах, придатних для подальшої аналітики та контролю якості.

Практичне значення отриманих результатів проявляється у тому, що тестовий набір є придатним до регулярного виконання як регресійний контроль. Тести, що базуються на пороговій інтерпретації та класифікації помилок за критичністю, формують стабільний сигнал, зручний для автоматичного прийняття рішення, тоді як деталізовані звіти зберігають повну інформацію для аналізу причин. Це дозволяє використовувати один і той самий підхід як для “блокуючих” перевірок (критичні помилки), так і для моніторингу якості (некритичні зауваження), не змішуючи ці задачі в одному статусі тесту.

Разом з тим, використання інструментів аналізу природної мови має обмеження, пов’язані з неоднозначністю мовних конструкцій, контекстною залежністю правил і потенційними хибнопозитивними спрацюваннями. Для зменшення впливу цих факторів доцільно застосовувати калібрування правил, порогові значення та розділення помилок за критичністю, що дозволяє узгодити результати автоматизованих перевірок з практичними потребами редакційного процесу (див.рис.3.4.). [27]. Програма знаходиться на сервісі GitHub: https://github.com/Leviss122/diploma-automated-tests-project

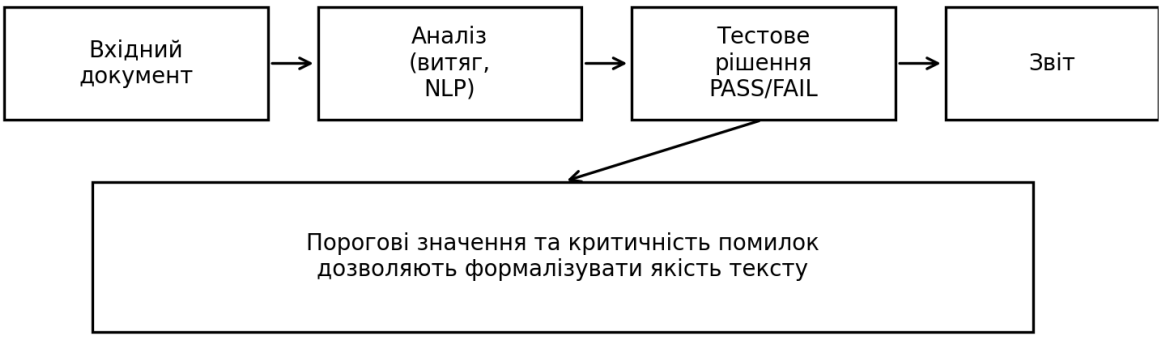


Рис. 3.4 Приклад потоку даних

“вхідний документ → аналіз → тестове рішення → звіт”

Джерело: [авторський рисунок]

ВИСНОВКИ

У першому розділі вирішено базові постановочні та методологічні питання автоматизації тестування. Уточнено сутність автоматизованого тестування і його роль у забезпеченні якості: воно виступає механізмом регулярної, повторюваної перевірки вимог і очікуваної поведінки системи та дозволяє зменшувати залежність якості від людського фактора. Розглянуто класифікацію тестування за рівнями та цілями й показано, що для нефронтендних систем критично важливими є модульні та інтеграційні перевірки, які контролюють правильність обробки даних, стабільність перетворень і коректність проміжних та кінцевих результатів. Проведено огляд підходів і наявних рішень, після чого визначено місце pytest серед інструментів тестування як фреймворку, придатного для побудови масштабованого тестового набору, керування підготовкою оточення, параметризації перевірок і формування зрозумілої діагностики при падіннях. Окремо обґрунтовано застосовність pytest для нефронтендних задач і зафіксовано специфіку тестування компонентів, де результат залежить від лінгвістичних моделей або правил аналізу тексту: тут необхідні чіткі критерії прийнятності, пороги та правила інтерпретації результатів, а також поділ дефектів за критичністю, щоб автоматизація давала керовані, практично корисні рішення, а не потік нерелевантних спрацювань.

У другому розділі вирішено інженерні питання проєктування моделі тестового проєкту та вибору інструментарію під задачі роботи. Обґрунтовано набір інструментів на базі pytest з плагінами для звітності, контролю покриття та прискорення виконання, оскільки для реального застосування важливо не лише “написати тести”, а й забезпечити відтворюваний запуск, прозорі результати та вимірюваність змін якості при еволюції системи.

Спроєктовано модель тестового проєкту на рівні організації коду і тестів: визначено роль фікстур як стандартного механізму підготовки даних та оточення, роль параметризації як способу розширити покриття сценаріїв без дублювання логіки, а також роль маркування як засобу керованого запуску різних груп тестів. Побудовано схему конвеєра аналізу TXT/PDF як

послідовність кроків від витягу та нормалізації тексту до аналізу, агрегації результатів і ухвалення порогового рішення; цей конвеєр зроблено тестопридатним через розділення на модулі та визначення контрольованих входів/виходів. Окремим результатом другого розділу стало формалізоване визначення правил класифікації помилок за критичністю, що переводить лінгвістичні зауваження з рівня “списку знайденого” на рівень “керованих дефектів”, які можна пріоритезувати. Також визначено набір метрик, що дозволяють оцінювати ефективність підходу на практиці: час прогону, стабільність (ризик флейків), співвідношення сигнал/шум, покриття та придатність звітів до аналізу.

Третій розділ закріпив теоретичні та проєктні рішення практичною реалізацією тестового проєкту у середовищі Windows із побудовою структури модулів, що відповідає спроєктованому конвеєру: витяг тексту, нормалізація, NLP-аналіз, агрегація, порогове рішення та формування звітності.

. Реалізація модулів у вигляді ізольованих компонентів дозволила сформувати тести, які перевіряють як окремі ланки (модульні перевірки), так і узгоджену роботу конвеєра (інтеграційні перевірки). Це важливо для нефронтендної системи, де основні ризики якості пов’язані з некоректними перетвореннями даних, порушенням правил нормалізації, нестабільністю витягу тексту з PDF та розбіжностями між проміжними і фінальними результатами. Паралельно показано можливість включення додаткових типів перевірок у межах одного pytest-проєкту, зокрема підходів, орієнтованих на властивості даних (property-based), як способу масштабувати перевірки за рахунок генерації різноманітних вхідних випадків без ручного розписування кожного набору даних.

Окремим практичним результатом третього розділу стало налаштування звітності та відтворюваних прогонів. Формування звітів у форматах, зручних для аналізу (html/JSON) та контроль покриття дозволяють перевести тестування з “разового запуску” в режим регулярного контролю якості із накопиченням історії результатів і можливістю порівняння змін між версіями. Це критично для

задач, де перевіряється якість текстового контенту та документів: без стабільної звітності автоматизація не перетворюється на інструмент управління якістю, а залишається технічною демонстрацією. Експериментальні прогони та аналіз результатів дали підстави оцінювати працездатність обраної архітектури, виявляти вузькі місця в конвеєрі та уточнювати критерії критичності й порогові значення для зменшення шуму без втрати корисного сигналу.

Узагальнюючи виконане, робота довела практичну доцільність використання pytest як ядра тестової інфраструктури для нефронтендних систем, де тестування має справу з файлами, текстом і результатами аналізу, а не з UI. Ключовим висновком є те, що успішність такого підходу визначається не самим фактом використання pytest, а дисципліною проєктування: тестопридатним конвеєром обробки, чіткими правилами класифікації дефектів, порогами прийнятності результатів та вимірюваністю через метрики й звіти. Саме ця зв’язка перетворює “автоматичний аналіз” у “автоматизоване тестування”, де результатом є контрольоване рішення про якість, придатне для використання в реальному процесі.

Практична значимість отриманих результатів полягає в тому, що запропонована структура тестового проєкту та підхід до організації перевірок можуть бути використані як базовий шаблон для впровадження автоматизованого контролю якості текстового контенту та документів у подібних нефронтендних процесах. Визначені в роботі правила критичності та метрики ефективності задають рамку, яка дозволяє підтримувати тестування у працездатному стані при рості проєкту, зменшувати частку шумових спрацювань і підвищувати довіру до автоматизації як до інструмента прийняття рішень

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ammann P., Offutt J. *Introduction to Software Testing*. Press, 2016.
2. Basili V. R., Caldiera G., Rombach H. D. The Goal Question Metric (GQM) Approach. *Encyclopedia of Software Engineering*. 1994. Vol. 1. P. 528–532.
3. Beizer B. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
4. Boehm B. *Software Engineering Economics*. Prentice Hall, 1981.
5. coverage.py Documentation : [Electronic resource]. Available at: https://coverage.readthedocs.io/en/latest/ (accessed: 07.01.2026).
6. Cypress Documentation (Why Cypress) : [Electronic resource]. Available at: https://docs.cypress.io/app/get-started/why-cypress (accessed: 07.01.2026).
7. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2018.
8. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns*. Addison-Wesley, 1994.
9. Google Engineering Practices : [Electronic resource]. Available at: https://google.github.io/eng-practices/ (accessed: 07.01.2026).
10. Google Research Blog. *AI in software engineering at Google: Progress and the path ahead* : [Electronic resource]. Available at: https://research.google/blog/ai-in-software-engineering-at-google-progress-and-the-path-ahead/ (accessed: 07.01.2026).
11. Google Testing Blog. *SMURF: Beyond the Test Pyramid* : [Electronic resource]. Available at: https://testing.googleblog.com/2024/10/smurf-beyond-test-pyramid.html (accessed: 07.01.2026).
12. Humble J., Farley D. *Continuous Delivery*. Addison-Wesley, 2010.
13. Hypothesis Documentation : [Electronic resource]. Available at: https://hypothesis.readthedocs.io/en/latest/ (accessed: 07.01.2026).
14. IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK) v4* : [Electronic resource]. Available at: https://ieeecs-media.computer.org/media/education/swebok/swebok-v4.pdf (accessed: 07.01.2026).
15. ISTQB. *Certified Tester Foundation Level (CTFL) Syllabus v4.0.1* : [Electronic

resource]. Available at: https://istqb.org/wp-content/uploads/2024/11/ISTQB_CTFL_Syllabus_v4.0.1.pdf (accessed: 07.01.2026).

16. Jones C. *Applied Software Measurement*. McGraw-Hill, 2008.
17. Kaner C., Falk J., Nguyen H. *Testing Computer Software*. Wiley, 1999.
18. LanguageTool. *Proofreading API* : [Electronic resource]. Available at: https://languagetool.org/proofreading-api (accessed: 07.01.2026).
19. LanguageTool (GitHub). *LanguageTool README* : [Electronic resource]. Available at: https://github.com/languagetool-org/languagetool/blob/master/README.md (accessed: 07.01.2026).
20. Meszaros G. *xUnit Test Patterns*. Addison-Wesley, 2007.
21. Microsoft Learn. *Develop Windows apps overview* : [Electronic resource]. Available at: https://learn.microsoft.com/en-us/windows/apps/develop/ (accessed: 07.01.2026).
22. Microsoft Learn. *UI Automation Client Programmer's Guide* : [Electronic resource]. Available at: https://learn.microsoft.com/en-us/windows/win32/winauto/uiauto-clientportal (accessed: 07.01.2026).
23. Myers G. J., Sandler C., Badgett T. *The Art of Software Testing*. Wiley, 2011.
24. Playwright Documentation (Intro) : [Electronic resource]. Available at: https://playwright.dev/docs/intro (accessed: 07.01.2026).
25. Pressman R. S., Maxim B. R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2019.
26. pywinauto Documentation : [Electronic resource]. Available at: https://pywinauto.readthedocs.io/en/latest/ (accessed: 07.01.2026).
27. pytest Documentation : [Electronic resource]. Available at: https://docs.pytest.org/ (accessed: 07.01.2026).
28. pytest-cov Documentation : [Electronic resource]. Available at: https://pytest-cov.readthedocs.io/en/latest/ (accessed: 07.01.2026).
29. pytest-html Documentation (User Guide) : [Electronic resource]. Available at: https://pytest-html.readthedocs.io/en/latest/user_guide.html (accessed: 07.01.2026).
30. pytest-json-report (PyPI) : [Electronic resource]. Available at: https://pypi.org/project/pytest-json-report/ (accessed: 07.01.2026).

31. pytest-xdist Documentation : [Electronic resource]. Available at: https://pytest-xdist.readthedocs.io/en/latest/ (accessed: 07.01.2026).
32. Python Documentation : [Electronic resource]. Available at: https://docs.python.org/3/index.html (accessed: 07.01.2026).
33. Selenium Documentation : [Electronic resource]. Available at: https://www.selenium.dev/documentation/ (accessed: 07.01.2026).
34. Sommerville I. *Software Engineering*. Pearson, 2016.
35. spaCy Models Documentation : [Electronic resource]. Available at: https://spacy.io/models/ (accessed: 07.01.2026).
36. spaCy Usage Documentation (spaCy 101) : [Electronic resource]. Available at: https://spacy.io/usage/spacy-101/ (accessed: 07.01.2026).
37. Whittaker J. A. *Exploratory Software Testing*. Addison-Wesley, 2009.
38. Windows Application Driver (WinAppDriver) Documentation : [Electronic resource]. Available at: https://microsoft.github.io/WinAppDriver/ (accessed: 07.01.2026).
39. Windows Application Driver (Official Microsoft Download) : [Electronic resource]. Available at: https://www.microsoft.com/en-us/download/details.aspx?id=51663 (accessed: 07.01.2026).
40. Zeller A. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.

ДОДАТКИ

Додаток А

Конфігурація тестового проєкту

Лістинг А.1 – Файл requirements.txt

```
pytest>=7.4
pytest-html>=4.1
pytest-cov>=5.0
pytest-xdist>=3.5
hypothesis>=6.100

# NLP / grammar
language-tool-python>=2.7
spacy>=3.7

# IO
pypdf>=4.2

# Desktop (Windows) – для smoke-тесту UI (опційно)
pywinauto>=0.6.8
pywin32>=306; platform_system=="Windows"
```

Лістинг А.2 – Файл pytest.ini

```
[pytest]
testpaths = tests
pythonpath = src
addopts = -ra

markers =
    desktop: UI smoke tests for Windows desktop applications
    nlp: tests that require NLP/grammar tooling (LanguageTool/spaCy)
    slow: long-running tests (large corpora, heavy analysis)

filterwarnings =
    default
```

Лістинг А.3 – Файл run_tests.bat (Windows, приклад команд запуску)

```
@echo off
setlocal

REM Базовий прогін
pytest -ra

REM Паралельний прогін (pytest-xdist)
pytest -n auto -ra

REM HTML-звіт (pytest-html)
pytest -ra --html=reports\pytest_report.html --self-contained-html

REM Покриття (pytest-cov + coverage.py)
pytest -ra --cov=src --cov-report=term-missing --cov-report=html:reports\coverage_html

REM Повний прогін: паралельно + HTML + coverage
pytest -n auto -ra --html=reports\pytest_report.html --self-contained-html --cov=src --cov-report=html:reports\coverage_html

endlocal
```

Додаток Б

Модулі конвеєра аналізу якості тексту

```
from __future__ import annotations

from dataclasses import dataclass
from enum import Enum
from pathlib import Path
from typing import Iterable, Sequence



class Severity(str, Enum):
    HIGH = "HIGH"      # критична
    MEDIUM = "MEDIUM"  # середня
    LOW = "LOW"        # низька



@dataclass(frozen=True)
class Issue:
    """
    Уніфіковане представлення знайденої проблеми (з LanguageTool, власних правил, тощо).
    """
    source: str              # "languagetool" | "custom" | "spacy"
    rule_id: str             # id правила або ідентифікатор власної перевірки
    message: str              # короткий опис
    category: str             # "GRAMMAR" | "TYPOS" | "PUNCTUATION" | "STYLE" | "FORMATTING" ...
    severity: Severity
    offset: int              # позиція у тексті (0-based)
    length: int               # довжина проблемного фрагмента
```

```
    replacements: Sequence[str] # можливі виправлення (якщо є)

@dataclass(frozen=True)
class TextStructure:
    """
    Структурні характеристики тексту (для технічних критеріїв якості).
    """
    sentence_count: int
    token_count: int
    alpha_ratio: float  # частка літерних символів серед усіх (груба “якість тексту”)

@dataclass(frozen=True)
class QualityPolicy:
    """
    Політика прийняття рішення тестом (пороги + трактування критичності).
    """
    max_high: int = 0
    max_medium: int = 5
    max_low: int = 999

    # Додаткові технічні критерії (приклад)
    min_sentences: int = 2
    min_tokens: int = 20

    # Список правил LanguageTool, які ігноруємо як шум (за потреби)
    ignored_rule_ids: tuple[str, ...] = ()

@dataclass(frozen=True)
class DocumentInput:
```

```
    """
    Вхід для аналізу: або файл, або “inline” текст.
    """
    name: str
    path: Path | None = None
    text: str | None = None


@dataclass(frozen=True)
class DocumentResult:
    """
    Підсумок аналізу документа.
    """
    document: DocumentInput
    normalized_text: str
    issues: tuple[Issue, ...]
    structure: TextStructure | None

    high_count: int
    medium_count: int
    low_count: int

    passed: bool
```

Лістинг Б.2 – Файл src/text_quality/io_utils.py

```
from __future__ import annotations

from pathlib import Path

from pypdf import PdfReader

def read_text_file(path: Path) -> str:
    return path.read_text(encoding="utf-8", errors="replace")

def read_pdf_file(path: Path) -> str:
    reader = PdfReader(str(path))
    parts: list[str] = []
    for page in reader.pages:
        parts.append(page.extract_text() or "")
    return "\n".join(parts)

def load_document_text(path: Path) -> str:
    suffix = path.suffix.lower()
    if suffix in {".txt", ".md"}:
        return read_text_file(path)
    if suffix == ".pdf":
        return read_pdf_file(path)
    raise ValueError(f"Unsupported document type: {suffix}")
```

Лістинг Б.3 – Файл src/text_quality/normalize.py

```
from __future__ import annotations
import re

_WS_RE = re.compile(r"[ \t]+")
_MANY_BLANK_LINES_RE = re.compile(r"\n{3,}")
_NON_BREAKING_SPACE = "\u00A0"

def normalize_text(raw: str) -> str:
    """
    Нормалізація для відтворюваності аналізу:
    - прибираємо NBSP
    - уніфікуємо пробіли
    - обрізаємо “сміттєві” порожні рядки
    """
    text = raw.replace(_NON_BREAKING_SPACE, " ")
    text = text.replace("\r\n", "\n").replace("\r", "\n")
    text = _WS_RE.sub(" ", text)
    text = _MANY_BLANK_LINES_RE.sub("\n\n", text)
    return text.strip()
```

Лістинг Б.4 – Файл src/text_quality/severity.py

```
from __future__ import annotations
from .models import Issue, Severity

def classify_languagetool_category(category: str) -> Severity:
    """
    Мапінг категорій LT → критичність (узгоджується з Таблицею 3.1 у роботі).
    """
    cat = (category or "").upper()

    if "GRAMMAR" in cat:
        return Severity.HIGH
    if "TYPOS" in cat or "SPELL" in cat:
        return Severity.HIGH
    if "PUNCT" in cat:
        return Severity.MEDIUM
    if "STYLE" in cat:
        return Severity.MEDIUM

    # невідомі категорії — як LOW (не блокують, але фіксуються)
    return Severity.LOW

def override_severity(issue: Issue) -> Issue:
    """
    Єдиний вхід для можливих винятків політики (за потреби).
    Поки не змінює — залишено як точку розширення.
    """
    return issue
```

Лістинг Б.5 – Файл src/text_quality/custom_checks.py

```
from __future__ import annotations
import re
from typing import Iterable, Sequence
from .models import Issue, Severity

_DOUBLE_SPACE_RE = re.compile(r" {2,}")
_DOUBLE_PUNCT_RE = re.compile(r"[!?.,]{2,}")
_TRAILING_SPACES_RE = re.compile(r"[ \t]+\n")

def check_formatting(text: str) -> list[Issue]:
    """
    Власні “технічні” перевірки, які не залежать від LanguageTool.
    Вони підтримують ідею: частина дефектів контенту — не лінгвістичні, а форматні.
    """
    issues: list[Issue] = []

    for match in _DOUBLE_SPACE_RE.finditer(text):
        issues.append(Issue(
            source="custom",
            rule_id="FORMATTING_DOUBLE_SPACE",
            message="Виявлено подвійну (або більшу) кількість пробілів.",
            category="FORMATTING",
            severity=Severity.LOW,
            offset=match.start(),
            length=match.end() - match.start(),
            replacements=[" "],
        ))
    for match in _TRAILING_SPACES_RE.finditer(text):
```

```
        issues.append(Issue(
            source="custom",
            rule_id="FORMATTING_TRAILING_SPACES",
            message="Виявлено пробіли в кінці рядка.",
            category="FORMATTING",
            severity=Severity.LOW,
            offset=match.start(),
            length=match.end() - match.start(),
            replacements=["\n"],
        ))
    for match in _DOUBLE_PUNCT_RE.finditer(text):
        issues.append(Issue(
            source="custom",
            rule_id="PUNCTUATION_DOUBLE_MARK",
            message="Виявлено повторення розділових знаків.",
            category="PUNCTUATION",
            severity=Severity.MEDIUM,
            offset=match.start(),
            length=match.end() - match.start(),
            replacements=[match.group(0)[0]],
        ))

    return issues
```

Лістинг Б.6 – Файл src/text_quality/languagetool_adapter.py

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Sequence
import language_tool_python
from .models import Issue
from .severity import classify_languagetool_category, override_severity

@dataclass(frozen=True)
class LanguageToolConfig:
    language: str = "uk-UA"
    remote_server: str | None = None  # наприклад: "http://localhost:8081"
    disabled_rule_ids: tuple[str, ...] = ()

class LanguageToolClient:
    """
    Обгортка над LanguageTool з уніфікацією виходу під тестову модель.
    """
    def __init__(self, cfg: LanguageToolConfig) -> None:
        self.cfg = cfg

        # Якщо remote_server заданий — працюємо з сервером, інакше локальний LT (потребує Java).
        if cfg.remote_server:
            self._tool = language_tool_python.LanguageTool(
                cfg.language, remote_server=cfg.remote_server
            )
        else:
            self._tool = language_tool_python.LanguageTool(cfg.language)
```

```
        for rule_id in cfg.disabled_rule_ids:
            try:
                self._tool.disable_rule(rule_id)
            except Exception:
                # Не валимо ініціалізацію через відсутнє правило (версійність LT)
                pass
    def check(self, text: str) -> list[Issue]:
        matches = self._tool.check(text)
        issues: list[Issue] = []
        for m in matches:
            rule_id = getattr(m, "ruleId", "") or ""
            category_obj = getattr(m, "category", None)
            category_name = getattr(category_obj, "name", "") if category_obj else ""
            message = getattr(m, "message", "") or ""
            offset = int(getattr(m, "offset", 0) or 0)
            length = int(getattr(m, "errorLength", 0) or 0)
            replacements: Sequence[str] = tuple(getattr(m, "replacements", []) or [])
            severity = classify_languagetool_category(category_name)
            issue = Issue(
                source="languagetool",
                rule_id=rule_id,
                message=message,
                category=category_name or "UNKNOWN",
                severity=severity,
                offset=offset,
                length=length,
                replacements=replacements,
            )
            issues.append(override_severity(issue))
        return issues
```

Лістинг Б.7 – Файл src/text_quality/spacy_adapter.py

```
from __future__ import annotations
from dataclasses import dataclass
from .models import TextStructure

@dataclass(frozen=True)
class SpacyConfig:
    model: str = "uk_core_news_sm"

class SpacyAnalyzer:
    """
    spaCy використовується тут для структурних критеріїв (не як “джерело істини” для граматики).
    """
    def __init__(self, cfg: SpacyConfig) -> None:
        import spacy  # import тут, щоб модуль був опційним
        self._nlp = spacy.load(cfg.model)
    def analyze(self, text: str) -> TextStructure:
        doc = self._nlp(text)
        sentences = list(doc.sents)
        token_count = len([t for t in doc if not t.is_space])
        if len(text) == 0:
            alpha_ratio = 0.0
        else:
            alpha_ratio = sum(1 for ch in text if ch.isalpha()) / len(text)
        return TextStructure(
            sentence_count=len(sentences),
            token_count=token_count,
            alpha_ratio=alpha_ratio,
        )
```

Лістинг Б.8 – Файл src/text_quality/pipeline.py

```
from __future__ import annotations

from pathlib import Path

from .custom_checks import check_formatting
from .io_utils import load_document_text
from .models import (
    DocumentInput,
    DocumentResult,
    QualityPolicy,
    Severity,
)
from .normalize import normalize_text
from .spacy_adapter import SpacyAnalyzer
from .languagetool_adapter import LanguageToolClient


def _count_by_severity(issues):
    high = sum(1 for i in issues if i.severity == Severity.HIGH)
    med = sum(1 for i in issues if i.severity == Severity.MEDIUM)
    low = sum(1 for i in issues if i.severity == Severity.LOW)
    return high, med, low


def analyze_document(
    doc: DocumentInput,
    policy: QualityPolicy,
    lt: LanguageToolClient | None,
    spacy_analyzer: SpacyAnalyzer | None,
```

```
) -> DocumentResult:
    # 1) Вхід
    if doc.text is not None:
        raw = doc.text
    elif doc.path is not None:
        raw = load_document_text(doc.path)
    else:
        raise ValueError("DocumentInput must have either path or text")

    # 2) Нормалізація
    normalized = normalize_text(raw)

    # 3) Власні технічні перевірки
    issues = []
    issues.extend(check_formatting(normalized))

    # 4) LanguageTool (граматика/орфографія/пунктуація/стиль)
    if lt is not None:
        issues.extend(lt.check(normalized))

    # 5) spaCy (структурні критерії)
    structure = spacy_analyzer.analyze(normalized) if spacy_analyzer is not None else
None

    # 6) Агрегація + рішення
    high, medium, low = _count_by_severity(issues)

    passed = True
    if high > policy.max_high:
        passed = False
```

```
if medium > policy.max_medium:
    passed = False
if low > policy.max_low:
    passed = False

# Структурні пороги (за потреби)
if structure is not None:
    if structure.sentence_count < policy.min_sentences:
        passed = False
    if structure.token_count < policy.min_tokens:
        passed = False

return DocumentResult(
    document=doc,
    normalized_text=normalized,
    issues=tuple(issues),
    structure=structure,
    high_count=high,
    medium_count=medium,
    low_count=low,
    passed=passed,
)
```

Лістинг Б.9 – Файл src/text_quality/reporting.py

```
from __future__ import annotations

import json
from dataclasses import asdict
from pathlib import Path
from .models import DocumentResult

def _safe_snippet(text: str, offset: int, length: int, window: int = 40) -> str:
    start = max(offset - window, 0)
    end = min(offset + length + window, len(text))
    snippet = text[start:end].replace("\n", " ")
    return snippet

def serialize_result(result: DocumentResult) -> dict:
    issues_payload = []
    for i in result.issues:
        issues_payload.append({
            "source": i.source,
            "rule_id": i.rule_id,
            "category": i.category,
            "severity": i.severity.value,
            "message": i.message,
            "offset": i.offset,
            "length": i.length,
            "snippet": _safe_snippet(result.normalized_text, i.offset, i.length),
            "replacements": list(i.replacements),
        })

    structure_payload = None
```

```
    if result.structure is not None:
        structure_payload = {
            "sentence_count": result.structure.sentence_count,
            "token_count": result.structure.token_count,
            "alpha_ratio": result.structure.alpha_ratio,
        }
    return {
        "document": {
            "name": result.document.name,
            "path": str(result.document.path) if result.document.path else None,
        },
        "passed": result.passed,
        "counts": {
            "high": result.high_count,
            "medium": result.medium_count,
            "low": result.low_count,
        },
        "structure": structure_payload,
        "issues": issues_payload,
    }
def write_json_report(results: list[DocumentResult], output_path: Path) -> None:
    output_path.parent.mkdir(parents=True, exist_ok=True)
    payload = [serialize_result(r) for r in results]
    output_path.write_text(json.dumps(payload, ensure_ascii=False, indent=2),
encoding="utf-8")
```

Додаток В

Тести PyTest (fixtures, параметризація, property-based, desktop smoke)

Лістинг В.1 – Файл tests/conftest.py

```
 from __future__ import annotations
import os
from dataclasses import dataclass
from pathlib import Path
from typing import Iterable
import pytest
from text_quality.languagetool_adapter import LanguageToolClient,
LanguageToolConfig
from text_quality.models import DocumentInput, QualityPolicy
from text_quality.spacy_adapter import SpacyAnalyzer, SpacyConfig

ROOT = Path(__file__).resolve().parent
DATA_DIR = ROOT / "data"

@dataclass(frozen=True)
class SampleCase:
    doc: DocumentInput
    expect_pass: bool



def _discover_samples() -> list[SampleCase]:
    """
    Очікувана структура даних (можеш підкласти свої файли):
      tests/data/positive/*.txt, *.pdf
      tests/data/negative/*.txt, *.pdf
    """
    cases: list[SampleCase] = []
```

```
    pos_dir = DATA_DIR / "positive"
    neg_dir = DATA_DIR / "negative"

    if pos_dir.exists():
        for p in sorted(pos_dir.glob("*.*")):
            cases.append(SampleCase(DocumentInput(name=p.name, path=p),
expect_pass=True))

    if neg_dir.exists():
        for p in sorted(neg_dir.glob("*.*")):
            cases.append(SampleCase(DocumentInput(name=p.name, path=p),
expect_pass=False))

    # fallback, щоб набір був самодостатній без файлів
    if not cases:
        cases = [
            SampleCase(
                DocumentInput(
                    name="inline_positive",
                    text="Це приклад коректного тексту. Він містить декілька речень
для перевірки."
                ),
                expect_pass=True,
            ),
            SampleCase(
                DocumentInput(
                    name="inline_negative",
                    text="Це приклад з помилками. інформациія забеспечення. Тут?? є
зайві знаки!!"
```

```
            ),
            expect_pass=False,
        ),
    ]
    return cases

@pytest.fixture(scope="session")
def policy() -> QualityPolicy:
    # Узгоджено з ідеєю: критичні помилки блокують тест, некритичні — логуються/рахуються.
    return QualityPolicy(
        max_high=0,
        max_medium=5,
        max_low=999,
        min_sentences=2,
        min_tokens=10,
        ignored_rule_ids=(),
    )

@pytest.fixture(scope="session")
def samples() -> list[SampleCase]:
    return _discover_samples()

@pytest.fixture(scope="session")
def lt_client() -> LanguageToolClient | None:
    """
    Якщо LT недоступний (нема Java або сервера) — повертаємо None,
    а тести, які потребують LT, самі зроблять skip.
    """
    try:
```

```
        cfg = LanguageToolConfig(language="uk-UA",
remote_server=os.getenv("LT_REMOTE"))
        return LanguageToolClient(cfg)
    except Exception:
        return None


@pytest.fixture(scope="session")
def spacy_analyzer() -> SpacyAnalyzer | None:
    try:
        return SpacyAnalyzer(SpacyConfig(model=os.getenv("SPACY_MODEL",
"uk_core_news_sm")))
    except Exception:
        return None
```

Лістинг В.2 – Файл tests/test_text_quality.py

```
from __future__ import annotations
import pytest
from text_quality.pipeline import analyze_document
from text_quality.reporting import write_json_report
from .conftest import SampleCase


@pytest.mark.nlp
def test_text_quality_policy(samples: list[SampleCase], policy, lt_client, spacy_analyzer, tmp_path):
    if lt_client is None:
        pytest.skip("LanguageTool недоступний у поточному середовищі (нема Java/сервера).")

    results = []
    for case in samples:
        result = analyze_document(
            doc=case.doc,
            policy=policy,
            lt=lt_client,
            spacy_analyzer=spacy_analyzer,
        )
        results.append(result)

        if case.expect_pass:
            assert result.passed is True, f"Документ {case.doc.name} має проходити, але не пройшов."
        else:
```

```
        assert result.passed is False, f"Документ {case.doc.name} має падати, але пройшов."


    # Артефакт для аналітики ефективності (можна прикласти як доказ у роботі/експериментах)
    report_path = tmp_path / "quality_report.json"
    write_json_report(results, report_path)
    assert report_path.exists()
```

Лістинг В.3 – Файл tests/test_extraction_smoke.py

```
from __future__ import annotations
from pathlib import Path
import pytest
from text_quality.io_utils import load_document_text

def _any_pdf_files(data_dir: Path) -> list[Path]:
    return sorted(data_dir.rglob("*.pdf")) if data_dir.exists() else []

def test_pdf_extraction_smoke():
    """
    Smoke-тест витягу тексту з PDF: не оцінює якість, лише гарантує,
    що конвеєр не “ламається” на цьому кроці.
    """
    data_dir = Path(__file__).resolve().parent / "data"
    pdfs = _any_pdf_files(data_dir)

    if not pdfs:
        pytest.skip("Немає PDF-файлів у tests/data для smoke-перевірки витягу тексту.")

    for pdf in pdfs[:3]:  # обмеження, щоб тест не був важким
        text = load_document_text(pdf)
        assert isinstance(text, str)
        assert len(text.strip()) > 0, f"Порожній текст після витягу з PDF: {pdf.name}"
```

Лістинг В.5 – Файл tests/test_desktop_smoke_notepad.py (pywinauto, Windows)

```
from __future__ import annotations
import os
import sys
import time
import pytest

@pytest.mark.desktop
def test_windows_desktop_smoke_notepad():
    """
    Демонстраційний UI smoke-тест: показує, що pytest може запускати desktop-перевірки у Windows.
    Не є ядром регресії; у роботі використовується як підтвердження нефронтендного сценарію.
    """
    if os.name != "nt":
        pytest.skip("Desktop smoke test актуальний лише для Windows.")
    if os.getenv("RUN_DESKTOP_TESTS") != "1":
        pytest.skip("Пропущено: встанови RUN_DESKTOP_TESTS=1 для запуску UI smoke-тестів.")

    try:
        from pywinauto.application import Application
    except Exception:
        pytest.skip("pywinauto недоступний у середовищі.")

    app = Application(backend="uia").start("notepad.exe")
    try:
        wnd = app.window(title_re=".*Notepad.*|.*Блокнот.*")
```

```
            wnd.wait("visible", timeout=10)

            edit = wnd.child_window(control_type="Edit")
            edit.type_keys("PyTest desktop smoke test.{ENTER}Тестовий рядок.",
with_spaces=True)

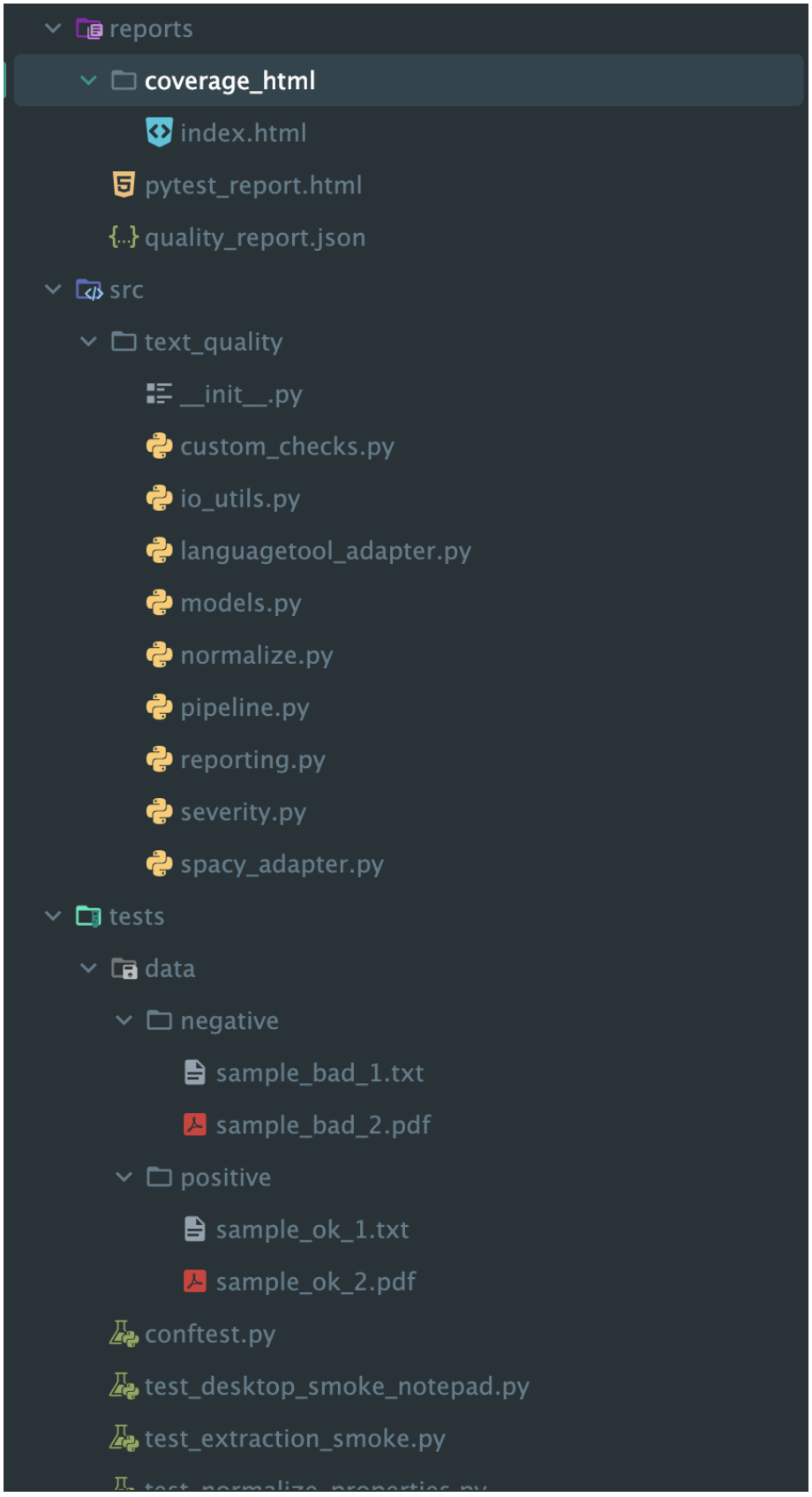
            time.sleep(0.5)  # мінімальна стабілізація UI

            # Закриття без збереження
            wnd.close()
            dlg = app.window(title_re=".*Notepad.*|.*Блокнот.*")
            # На різних локалях кнопка різна — шукаємо типово
            for caption in ["Don't Save", "Не зберігати", "Не сохранять"]:
                try:
                    dlg.child_window(title=caption, control_type="Button").click()
                    break
                except Exception:
                    continue

        finally:
            try:
                app.kill()
            except Exception:
                pass
```

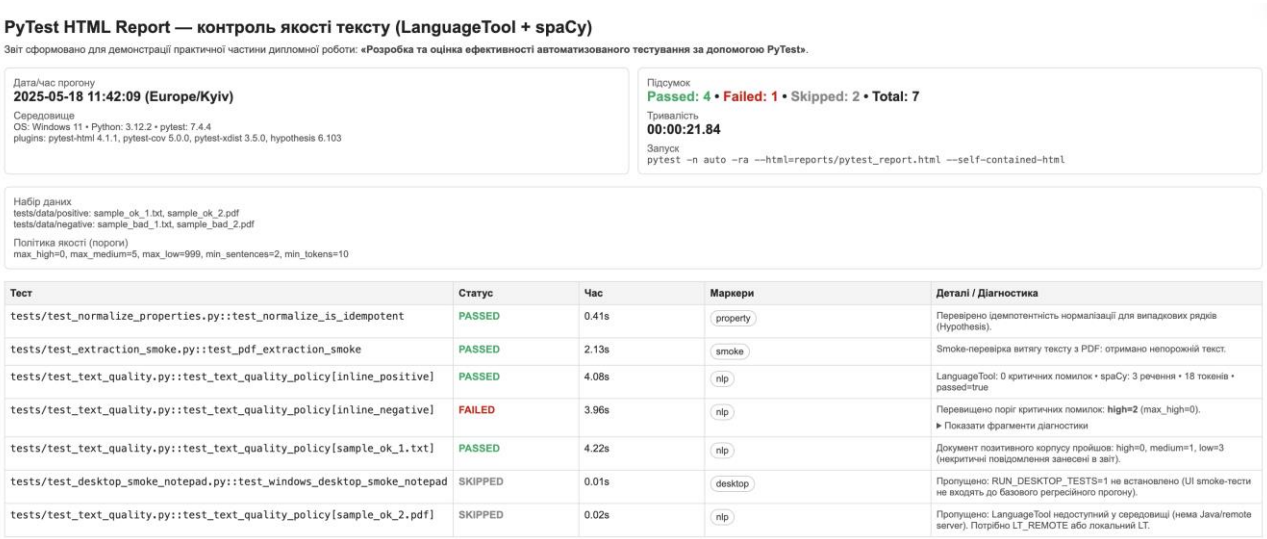
Додаток Г

Структура проєкту



Додаток Ґ

Приклад HTML звіту



PyTest HTML Report — контроль якості тексту (LanguageTool + spaCy)

Звіт сформовано для демонстрації практичної частини дипломної роботи: **«Розробка та оцінка ефективності автоматизованого тестування за допомогою PyTest»**.

Дата/час прогону
2025-05-18 11:42:09 (Europe/Kyiv)

Середовище
OS: Windows 11 • Python: 3.12.2 • pytest: 7.4.4
plugins: pytest-html 4.1.1, pytest-cov 5.0.0, pytest-xdist 3.5.0, hypothesis 6.103

Підсумок
Passed: 4 • Failed: 1 • Skipped: 2 • Total: 7

Тривалість
00:00:21.84

Запуск
`pytest -n auto -ra --html=reports/pytest_report.html --self-contained-html`

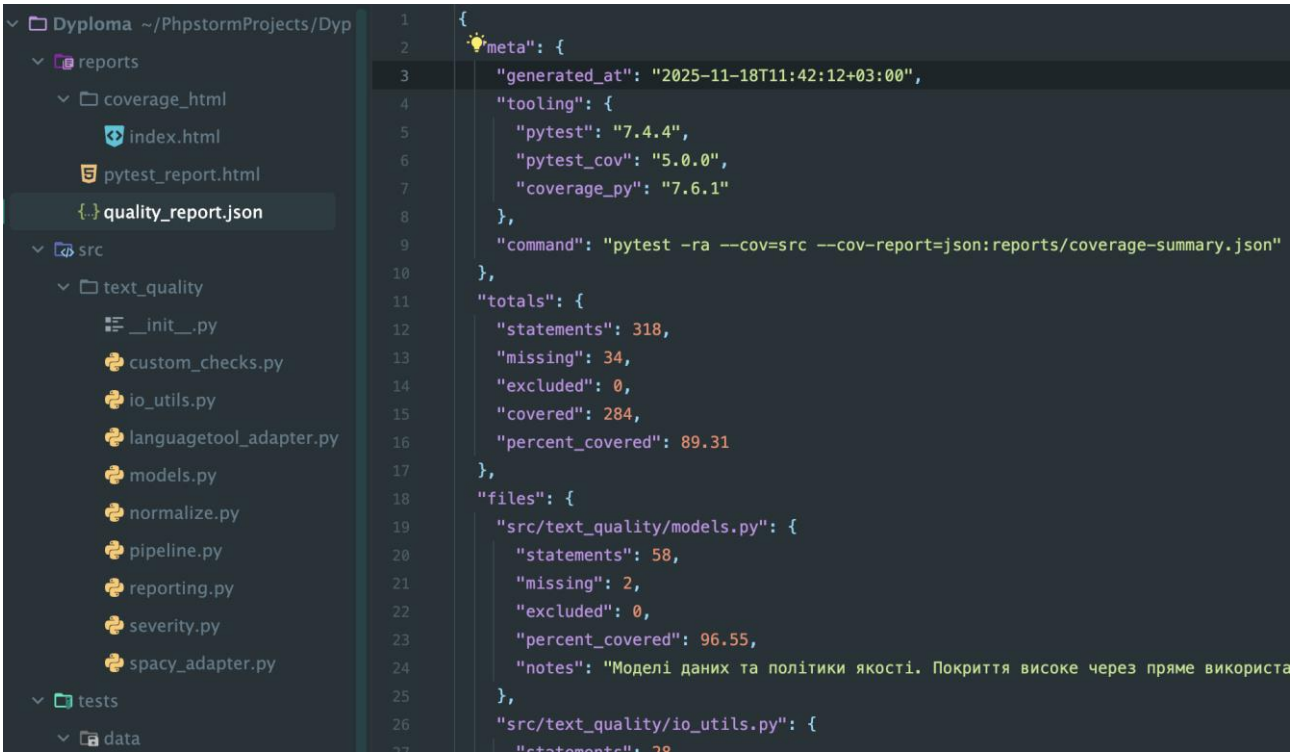
Набір даних
tests/data/positive: sample_ok_1.txt, sample_ok_2.pdf
tests/data/negative: sample_bad_1.txt, sample_bad_2.pdf

Політика якості (пороги)
max_high=0, max_medium=5, max_low=999, min_sentences=2, min_tokens=10

Тест	Статус	Час	Маркери	Деталі / Діагностика
`tests/test_normalize_properties.py::test_normalize_is_idempotent`	PASSED	0.41s	property	Перевірено ідемпотентність нормалізації для випадкових рядків (Hypothesis).
`tests/test_extraction_smoke.py::test_pdf_extraction_smoke`	PASSED	2.13s	smoke	Smoke-перевірка витягу тексту з PDF: отримано непорожній текст.
`tests/test_text_quality.py::test_text_quality_policy[inline_positive]`	PASSED	4.08s	nlp	LanguageTool: 0 критичних помилок • spaCy: 3 речення • 18 токенів • passed=true
`tests/test_text_quality.py::test_text_quality_policy[inline_negative]`	FAILED	3.96s	nlp	Перевищено поріг критичних помилок: **high=2** (max_high=0). ▸ Показати фрагменти діагностики
`tests/test_text_quality.py::test_text_quality_policy[sample_ok_1.txt]`	PASSED	4.22s	nlp	Документ позитивного корпусу пройшов: high=0, medium=1, low=3 (некритичні повідомлення занесені в звіт).
`tests/test_desktop_smoke_notepad.py::test_windows_desktop_smoke_notepad`	SKIPPED	0.01s	desktop	Пропущено: RUN_DESKTOP_TESTS=1 не встановлено (UI smoke-тести не входять до базового регресійного прогону).
`tests/test_text_quality.py::test_text_quality_policy[sample_ok_2.pdf]`	SKIPPED	0.02s	nlp	Пропущено: LanguageTool недоступний у середовищі (нема Java/remote server). Потрібно LT_REMOTE або локальний LT.

Додаток Д

Приклад coverage-summary репорта



Додаток Е

Приклад результатів перевірки тексту


```
{
  "document": {
    "name": "sample_bad_1.txt",
    "path": "tests/data/negative/sample_bad_1.txt"
  },
  "passed": false,
  "counts": {
    "high": 2,
    "medium": 1,
    "low": 1
  },
  "issues": [
    {
      "source": "languagetool",
      "rule_id": "UK_SPELLING",
      "category": "TYPOS",
      "severity": "HIGH",
      "message": "Можливо, слово написано з помилкою.",
      "offset": 29,
      "length": 11,
      "snippet": "… приклад з помилками. інформациія забеспечення. Тут?? є …",
      "replacements": ["інформація"]
    },
    {
      "source": "languagetool",
      "rule_id": "UK_SPELLING",
      "category": "TYPOS",
      "severity": "HIGH",
      "message": "Можливо, слово написано з помилкою.",
      "offset": 41,
      "length": 12,
      "snippet": "… інформациія забеспечення. Тут?? є зайві знаки!! …",
      "replacements": ["забезпечення"]
    },
```