КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій проектування та прикладної математики

(кафедра)

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ

НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Методи тестування програмного забезпечення та оцінка якості його функціонування»

ГОЙКО ФРАНЦ ОЛЕКСАНДРОВИЧ
(прізвище, ім’я та по батькові студента повністю)

Київ 2026 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій проектування та прикладної математики

(кафедра)

ЗАТВЕРДЖУЮ

Завідувач кафедри ІТППМ

д.т.н., професор Бородавка Є.В.

_____

„___” _____2026 року

ПОЯСНЮВАЛЬНА ЗАПИСКА

ДО КВАЛІФІКАЦІЙНОЇ РОБОТИ

НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Методи тестування програмного забезпечення та оцінка якості його функціонування»

Виконав: студент 4-го курсу, групи ІСТ-22

Спеціальності: 126 «Інформаційні системи та технології»

(шифр і назва спеціальності)

Гойко Ф.О.

(прізвище та ініціали)

Керівник д.т.н., професор Терентьєв О.О.

(прізвище та ініціали)

Рецензент

(прізвище та ініціали)

Київ, 2026 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій
Кафедра: інформаційних технологій проєктування та ПМ
Освітній рівень: «бакалавр» за ОПП
Спеціальність: 126 «Інформаційні системи та технології»
Освітня програма: «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТППМ
д.т.н., професор Бородавка Є.В.
_____
„___” _____2026 року

З А В Д А Н Н Я
ДО ВИКОНАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»
Гойко Франц Олександрович

1.Тема роботи: Методи тестування програмного забезпечення та оцінка якості його функціонування затверджена наказом ректора КНУБА № 444/52-15/13.6/26 від “08” квітня 2026 року
2. Керівник роботи: Терентьєв Олександр Олександрович, д.т.н, професор кафедри інформаційних технологій проектування і прикладної математики
3. Строк подання студентом роботи до захисту: червень 2026 року
4. Зміст пояснювальної записки за розділами:
Р.1. Аналіз предметної області методів тестування програмного забезпечення та оцінки його якості функціонування
Р.2. Рівні тестування програмного забезпечення для оцінки якості його функціонування
Р.3. Практична реалізація тестування програмного забезпечення
Р.4. Аналіз методів тестування та оцінка якості програмного забезпечення
5. Інформаційні слайди:
С.1.Актуальність та мета
С.2. Об’єкт та предмет дослідження
С.3. Рівні тестування та методи тестування

С.4. Аналіз виявлених дефектів
С.5 Оцінка якості програмного забезпечення
С.6. Висновки

6. Календарний план виконання кваліфікаційної роботи бакалавра

Види робіт та їх зміст	Дата виконання
Р. 1. Аналіз предметної області методів тестування програмного забезпечення та оцінки його якості функціонування	Лютий 2026 р.
Р. 2. Рівні тестування програмного забезпечення для оцінки якості його функціонування	Квітень 2026 р.
Р. 3. Практична реалізація тестування програмного забезпечення	Квітень 2026 р.
Р. 4. Аналіз методів тестування та оцінка якості програмного забезпечення	Травень 2026 р.
Остаточне оформлення роботи	Травень 2026 р.
Направлення роботи на рецензування	Червень 2026 р.
Попередній захист роботи на кафедрі	Червень 2026 р.

7. Консультанти розділів кваліфікаційної роботи бакалавра

Розділ	Прізвище, ініціали та посада консультанта, представника комісії	дата	підпис
Прийом програмного продукту	д.т.н., професор Бородавка Є.В.		

8. Дата видачі завдання: 12 січня 2026 р.

Керівник _____ Терентьєв О.О.
(підпис) (прізвище та ініціали)

Бакалавр _____ Гойко Ф.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

«Методи тестування програмного забезпечення та оцінка якості його функціонування».

Кваліфікаційна робота бакалавра за спеціальністю: 126 «Інформаційні системи та технології», освітня програма: «Інформаційні системи та технології». – Київський національний університет будівництва та архітектури. – Київ, 2026.

Робота присвячена дослідженню методів тестування програмного забезпечення та практичній реалізації комплексного підходу до оцінки якості його функціонування. У роботі розглядаються методи «чорної», «білої» та «сірої» скриньок, рівні тестування, а також сучасні інструменти автоматизації перевірки програмних продуктів. За результатами роботи розроблено тест-план, реалізовано набори функціональних та автоматизованих тестів, проведено оцінку якості програмного забезпечення.

Ключові слова: тестування програмного забезпечення, якість програмного забезпечення, функціональне тестування, модульне тестування, інтеграційне тестування, покриття коду, метрики якості.

SUMMARY

" Software testing methods and evaluation of the quality of its functioning ".

Bachelor's qualification work in the specialty: 126 "Information systems and technologies", educational program: "Information systems and technologies". - Kyiv National University of Construction and Architecture. - Kyiv, 2026.

The work is devoted to the research of software testing methods and the practical implementation of a comprehensive approach to assessing the quality of its functioning. The work examines the black-box, white-box and grey-box testing methods, levels of testing, as well as modern tools for automated verification of software products. The practical part is implemented on the example of the computer game "Cosmic Heat", developed in Python. Based on the results of the work, a test

plan was developed, sets of functional and automated tests were implemented, and the software quality.

Keywords: software testing, software quality, functional testing, unit testing, integration testing, code coverage, quality metrics

ЗМІСТ

ВСТУП 1

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ МЕТОДІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОЦІНКИ ЯКОСТІ ЙОГО ФУНКЦІОНУВАННЯ 3

1.1 Опис предметної області 3

1.2 Дослідження методів тестування програмного забезпечення 10

1.3 Порівняльний аналіз підходів тестування та оцінки якості програмного забезпечення 17

1.4 Постановка задачі 20

2. РІВНІ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОЦІНКИ ЯКОСТІ ЙОГО ФУНКЦІОНУВАННЯ 22

2.1 Рівні тестування програмного забезпечення 22

2.2 Модульний рівень тестування 23

2.3 Інтеграційний рівень тестування 30

2.4 Системний рівень тестування 34

2.5 Приймальний рівень тестування 38

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 41

3.1 Опис об’єкта тестування 41

3.2 Розробка тест-плану 45

3.3 Тестування методом «чорної скриньки» 48

3.4 Тестування методом «білої скриньки» 51

3.5 Тестування методом «сірої скриньки» 56

4. АНАЛІЗ МЕТОДІВ ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 61

4.1 Порівняльний аналіз застосованих методів тестування 61

4.2 Аналіз виявлених дефектів 67

4.3 Оцінка якості програмного забезпечення 74

4.4 Рекомендації щодо покращення програмного забезпечення 78

ВИСНОВКИ 80

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

ВСТУП

У сучасному світі програмне забезпечення стало невід'ємною частиною повсякденного життя. Мобільні додатки, веб-платформи, ігрові застосунки та корпоративні системи – усе це вимагає не лише якісної розробки, а й ретельної перевірки перед введенням в експлуатацію. Зі зростанням складності програмних продуктів зростають і вимоги до їх якості. Будь-яка помилка у програмному забезпеченні може призвести до некоректної роботи системи або незадоволення користувачів. Саме тому тестування програмного забезпечення є одним із найважливіших етапів його розробки.

Актуальність теми зумовлена стрімким зростанням складності сучасних програмних систем та підвищенням вимог до їх якості. В умовах активного розвитку інформаційних технологій надійність програмного забезпечення стає критично важливим фактором, оскільки збій у роботі програмного продукту може призводити до серйозних наслідків.

Проте на практиці, особливо у невеликих та навчальних проєктах, тестуванню приділяється недостатньо уваги. Відсутність систематичного підходу до перевірки якості призводить до того, що дефекти виявляються вже після випуску продукту, що значно ускладнює їх усунення та підвищує витрати на підтримку. Вирішенням цієї проблеми є застосування сучасних методів тестування: функціонального, модульного та інтеграційного – у поєднанні з інструментами автоматизації. Такий підхід дозволяє підвищити ефективність перевірки, зменшити вплив людського фактору та забезпечити об'єктивну оцінку якості програмного продукту.

Метою даної роботи є дослідження методів тестування програмного забезпечення, та їх практична реалізація на прикладі прикладного програмного забезпечення. У роботі розглядаються підходи «чорної», «білої» та «сірої» скриньок, рівні тестування, автоматизовані тести з використанням фреймворку pytest, а також оцінка якості за міжнародним стандартом ISO/IEC 25010.

Об'єктом дослідження є прикладне програмне забезпечення «Cosmic Heat» – гра, розроблена мовою Python з використанням бібліотеки pygame.

Предметом дослідження є методи тестування програмного забезпечення «чорної», «білої» та «сірої» скриньок, рівні тестування та метрики оцінки якості за стандартом ISO/IEC 25010.

Основними методами дослідження є:

- Аналіз та систематизація існуючих підходів до тестування програмного забезпечення.
- Порівняльний аналіз методів тестування за ступенем автоматизації.
- Функціональне тестування методом «чорної скриньки» для перевірки ігрової механіки застосунку.
- Модульне тестування методом «білої скриньки» з використанням фреймворку pytest.
- Інтеграційне тестування із застосуванням системи логування подій.
- Кількісна оцінка якості програмного забезпечення на основі метрик покриття коду та щільності дефектів.
- Оцінювання якості за міжнародним стандартом ISO/IEC 25010.

Розроблена методика тестування може бути застосована для перевірки якості інших програмних застосунків з об'єктно-орієнтованою архітектурою. Отримані результати демонструють ефективність комплексного використання різних методів тестування та можуть слугувати практичним прикладом для розробників під час організації процесу забезпечення якості програмного забезпечення.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ МЕТОДІВ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ОЦІНКИ ЯКОСТІ ЙОГО ФУНКЦІОНУВАННЯ

1.1 Опис предметної області

У сучасному світі інформаційних технологій, програмне забезпечення відіграє ключовий етап функціонування майже всіх сфер людської діяльності. Програмні забезпечення, електронні сервіси, мобільні додатки та веб-платформи вимагають високого рівня надійності, безпеки, стабільності та продуктивності. Зі зростанням складності програмних продуктів зростають і вимоги до їх якості, що у свою чергу зумовлює необхідність застосування ефективних методів тестування та оцінки працездатності програмного забезпечення.

Якість програмного забезпечення визначається сукупністю характеристик, які відображають його відповідність встановленим вимогам і очікуванням користувачів. За даними дослідження Consortium for Information & Software Quality, проблеми, пов’язані зі якістю програмного забезпечення, щороку завдають економіці США збитків на суму понад 2,08 трильйона доларів. Цей факт підкреслює критичну важливість забезпечення високої якості розроблюваних продуктів. Недостатня якість може призвести до фінансових втрат, зниження довіри користувачів, ризиків для безпеки даних та збоїв у функціонуванні критично важливих систем. Тож тестування програмного забезпечення є невід’ємною частиною життєвого циклу його розробки.

Тестування програмного забезпечення – це процес перевірки та оцінки коректності програмного продукту з метою виявлення помилок, дефектів та невідповідностей встановленим вимогам.

Основною метою тестування є забезпечення належного рівня якості функціонування системи перед її впровадженням або введенням в експлуатацію. Тестування дозволяє мінімізувати ризики, пов’язані з експлуатацією програмного продукту, та забезпечити його стабільну роботу в реальних умовах.

Згідно зі звітом World Quality Report (Рис.1.1), впровадження цілісного підходу до тестування та автоматизації значно впливає на покращення результатів процесу розробки:

- **64%** компаній – демонструють про підвищення своєчасності релізів.
- **62%** компаній – зниження витрат на забезпечення якості.
- **61%** компаній – покращення користувацького досвіду.

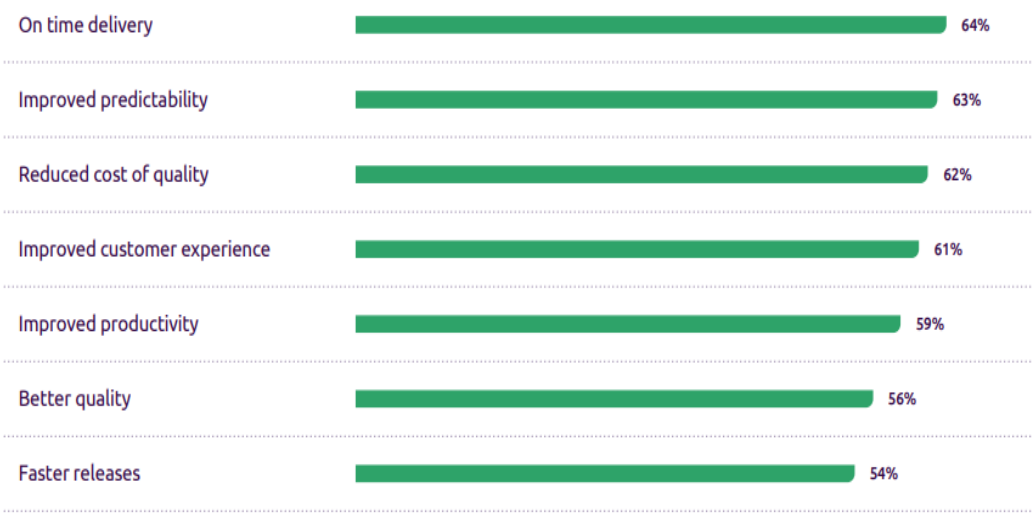


Рис.1.1 Звіт World Quality Report

Це демонструє, що впровадження системної інтеграції тестування прямо впливає на покращення якості програмного забезпечення і сприяє зниженню кількості дефектів у продуктивному середовищі.

Сфера дослідження у тестуванні програмних продуктів включає широкий спектр методів і підходів, які застосовуються залежно від типу системи, етапу розробки та поставлених вимог. Вона включає:

Функціональне тестування – спрямоване на перевірку відповідності реалізованого функціоналу технічному завданню.

Нефункціональне тестування – воно оцінює характеристики продуктивності, надійності, безпеки та зручності використання.

Для представлення використання рівнів тестування програмного забезпечення наведено у в таблиці 1.1 їх основні цілі та відповідні інструменти реалізації.

Таблиця 1.1

Використання рівнів тестування програмного забезпечення

Рівень тестування	**Функціональне**	**Нефункціональне**
Unit testing	Перевіряє правильність вхідних і вихідних даних для функції	Вимірюють швидкість виконання окремих алгоритмів
Integration testing	Чи правильно передаються дані між компонентами	Перевірка швидкості взаємодії та безпеки з'єднань
System testing	Перевірка всіх бізнес-вимог та сценаріїв використання	Основний етап для таких тестів, як тестування продуктивності (Performance), безпеки (Security), зручності використання (Usability)
Acceptance testing	Чи відповідає кінцевий продукт очікуванням бізнесу	Зручність, відповідність стандартам доступності

Вибір конкретного інструменту залежить від типу системи та етапу тестування.

Процес тестування зазвичай складається з кількох взаємопов’язаних етапів. Відповідно до стандарту IEEE 29119, процес тестування програмного забезпечення супроводжується документацією, яка охоплює планування, проєктування, виконання та аналіз результатів тестування таких як:

- Спочатку потрібно дізнатися аналіз вимог, після чого сформувати тест-план.
- Виконується підготовка тестового середовища та необхідних даних.
- Наступним етапом є безпосереднє виконання тестів, передача об’єктів фіксація виявлених дефектів та їхній подальший аналіз.
- Завершальним етапом є оцінка отриманих результатів і прийняття рішення щодо готовності програмного продукту до релізу.

Логіку виконання процесу тестування програмного забезпечення наведено на Рис. 1.2, що відображає послідовність ключових етапів – від аналізу вимог, до прийняття рішення щодо готовності продукту до релізу.

Рис. 1.2 Процес тестування програмного забезпечення

Особливістю сучасного тестування є активне використання автоматизованих засобів, які дозволяють підвищити ефективність перевірки програмних систем.

Сучасний процес тестування програмного забезпечення базується на використанні спеціалізованих інструментів. Використання таких інструментів дозволяє скоротити час перевірки та підвищити об’єктивність оцінювання результатів. Оцінка якості функціонування програмного забезпечення базується

на визначених критеріях та метриках. Однією з найбільш поширених моделей якості є стандарт ISO/IEC 25010 (Рис. 1.3).

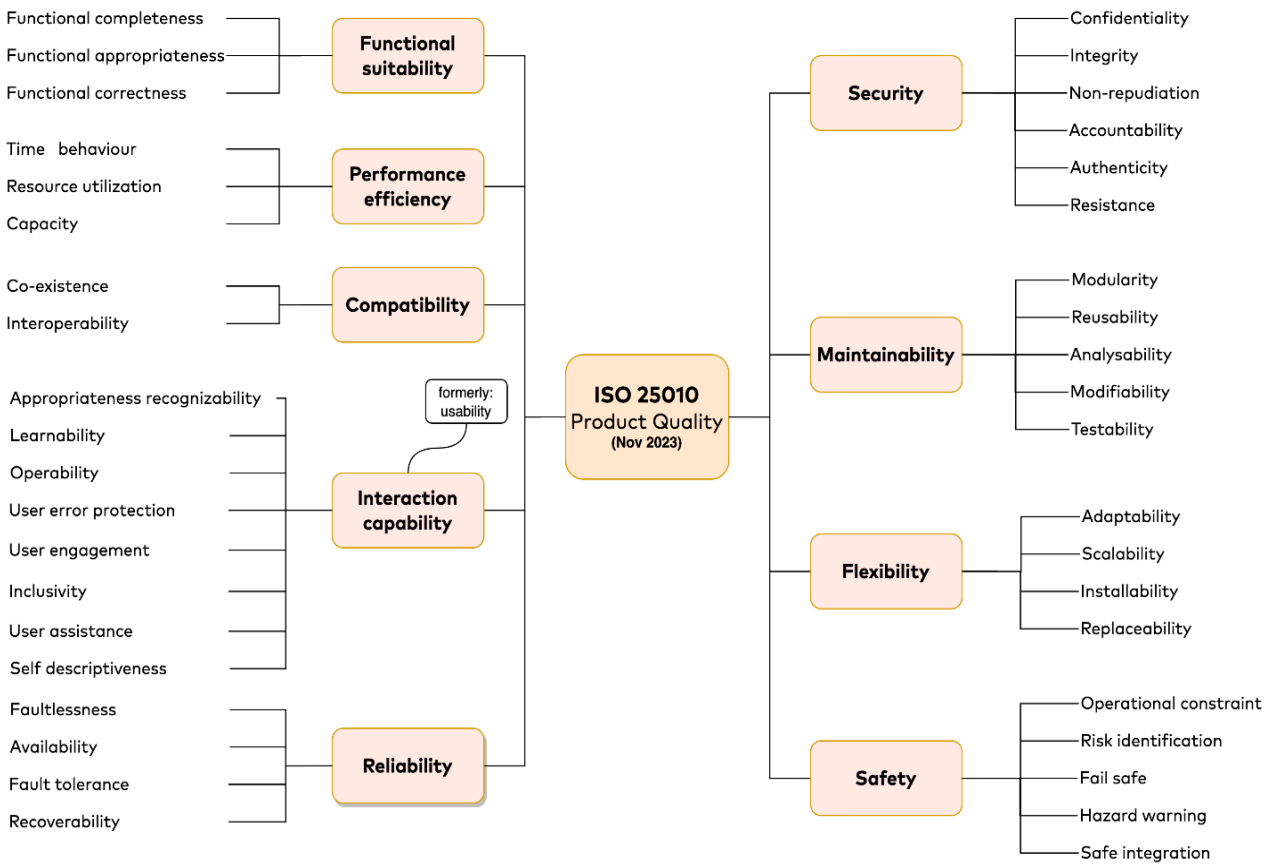


Рис. 1.3 Модель якості ISO/IEC 25010

Згідно з цим стандартом, якість програмного забезпечення оцінюється за такими основними характеристиками:

Надійність (Reliability) – здатність системи стабільно виконувати функції протягом визначеного часу.

Функціональна придатність (Functional Suitability) – здатність програмного забезпечення забезпечувати необхідні функції відповідно до встановлених вимог.

Продуктивність (Performance Efficiency) – здатність системи ефективно виконувати функції з оптимальним використанням ресурсів.

Сумісність (Compatibility) – здатність програмного продукту взаємодіяти з іншими системами або компонентами в спільному середовищі.

Безпека (Safety) – здатність захищати дані та забезпечувати контроль доступу до них.

Гнучкість (Flexibility) – здатність системи адаптуватися до зміни середовища або вимог.

Взаємодія (Interaction Capability) – рівень зручності та ефективності взаємодії користувача з системою.

Підтримуваність (Maintainability) – здатність системи стабільно виконувати свої функції протягом конкретного часу.

Використання стандартизованих підходів до оцінювання дозволяє забезпечити системність та об’єктивність аналізу якості.

Предметна область тестування програмного забезпечення та оцінки його функціональної якості охоплює широкий спектр питань, що поєднують знання з програмної інженерії, методів забезпечення якості та сучасних технологій автоматизації. У сучасних умовах розробки складних інформаційних систем тестування вже не обмежується лише пошуком помилок, а воно включає оцінку надійності, продуктивності та відповідності програмного забезпечення потребам користувача.

Особливо актуальним стає використання автоматизованих та інтелектуальних підходи тестування, здатних обробляти великі обсяги даних і враховувати різноманітні сценарії використання програмних продуктів.

Дослідження arXiv та IBM стверджують, що застосування штучного інтелекту в тестуванні дозволяє підвищити ефективність виявлення дефектів та автоматизувати генерацію тест-кейсів, що значно зменшує витрати часу на перевірку.

Такі підходи дозволяють не лише скоротити час і ресурси на перевірку програм, але й підвищити точність знаходження критичних дефектів, мінімізуючи вплив людського фактору.

Враховуючи швидкі темпи розвитку програмного забезпечення та збільшення складності систем, дослідження сучасних методів тестування та розробка ефективних інструментів оцінки якості стають надзвичайно актуальними. Впровадження таких рішень забезпечує підвищення надійності продуктів, їхню безперервну підтримку та масштабованість у процесі експлуатації, що робить цю область дослідження перспективною та значущою для практики розробки програмного забезпечення.

1.2 Дослідження методів тестування програмного забезпечення

Класифікація методів тестування програмного забезпечення

Методи тестування застосовують за такими класифікаціями як: «чорної скриньки», «білої скриньки» та «сірої скриньки» (Рис.1.4).

Метод «чорної скриньки» передбачає перевірку функціональності системи без аналізу її внутрішньої структури. Тестування здійснюється за принципом вхідних та вихідних даних, що дозволяє оцінити відповідність програмного продукту вимогам користувача.

Натомість метод «білої скриньки» базується на аналізі внутрішньої логіки програми, структури коду та алгоритмів.

Метод «сірої скриньки» поєднує елементи обох підходів, забезпечуючи більш комплексну перевірку. Як стверджує автор Борис Бейзер у своїй роботі, що використання методів «білої скриньки» може забезпечити покриття до 80-90% гілок виконання коду, тоді як «чорна скринька» орієнтована на перевірку відповідності вимогам.

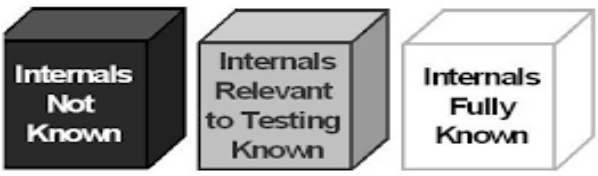


Рис.1.4 Методи «скриньок»

Класичні методи тестування програмного забезпечення за ступенем автоматизації

Класичні методи тестування програмного забезпечення за ступенем автоматизації – є фундаментальним методом перевірки якості програмних продуктів, що сформувався на ранніх етапах розвитку програмної інженерії. Вони ґрунтуються на формалізованих правилах, сценаріях перевірки та аналізі відповідності реалізованого функціоналу встановленим вимогам. Незважаючи на появу автоматизованих і інтелектуальних методів, класичні методи залишаються важливою складовою процесу забезпечення якості.

Одним із найбільш поширених класичних методів – є ручне тестування. Його суть полягає у безпосередньому виконанні тестових сценаріїв тестувальником без використання спеціалізованих засобів автоматизації. Перевірка здійснюється відповідно до підготовлених тест-кейсів або на основі досвіду спеціаліста. За даними звіту Capgemini, попри активний розвиток автоматизації, близько 30-40% тестових активностей у проєктах все ще виконуються вручну, особливо на етапах дослідницького (exploratory) тестування.

Перевага класичних методів полягає у структурованості та передбачуваності. Вони дозволяють систематизувати процес перевірки, забезпечити документування результатів та відстеження дефектів. Крім того, такі підходи не потребують складної технічної інфраструктури та можуть застосовуватися на будь-якому етапі життєвого циклу розробки.

Водночас класичні методи мають певні обмеження. Ручне тестування є трудомістким і потребує значних часових витрат. Із зростанням складності програмних систем збільшується кількість тестових сценаріїв, що ускладнює їх повне покриття. Крім того, людський фактор може впливати на об’єктивність результатів, оскільки різні тестувальники можуть по-різному інтерпретувати вимоги або сценарії перевірки. Ще одним недоліком є складність масштабування

класичних методів у проєктах із частими релізами та безперервною інтеграцією. У таких умовах повторне виконання великої кількості тестів вручну стає неефективним, що зумовлює необхідність впровадження автоматизованих засобів контролю якості.

Обмеження сприяли активному розвитку автоматизованих методів тестування, які дозволяють підвищити швидкість, точність та масштабованість процесу перевірки програмних продуктів.

Автоматизовані методи тестування програмного забезпечення за ступенем автоматизації

Автоматизовані методи тестування програмного забезпечення за ступенем автоматизації, передбачають підхід використання спеціалізованих інструментів та програмних платформ для виконання тестових сценаріїв без безпосередньої участі людини.

Головна ідея такого методу полягає у формалізації повторюваних дій та їх автоматичному виконанні, що дозволяє підвищити ефективність перевірки, зменшити часові витрати та забезпечити більш об’єктивну оцінку результатів.

Види тестування, що можуть бути реалізовані засобами автоматизації, належать:

- **Системне (System testing);**
- **Модульне (Unit testing);**
- **Інтеграційне (Integration testing);**
- **Регресійне (Regression testing);**

Їхнє призначення полягає у тому що:

- **Системне тестування** – може виконуватись для перевірки роботи програмного продукту у цілому.
- **Модульне тестування** – передбачає перевірку окремих компонентів або функцій системи, зазвичай за допомогою автоматизованих тестів.
- **Інтеграційне тестування** – спрямоване на оцінку коректності взаємодії між модулями системи.

- **Регресійне тестування** – використовується для виявлення помилок, що могли з’явитися після внесення змін у програмний код.

Перевагою методів автоматизованого тестування є його повторюваність та об’єктивність. Автоматизовані тести дозволяють виконувати однакові сценарії багаторазово, без ризику людської помилки, що особливо важливо при великих проєктах із частими оновленнями. Крім того, автоматизація сприяє інтеграції тестування у цикли безперервної розробки (CI/CD), забезпечуючи постійний контроль якості на всіх етапах життєвого циклу продукту.

Разом із тим, автоматизовані методи тестування мають певні обмеження. Розробка автоматизованих тестів потребує значних початкових зусиль та спеціалізованих знань з мови програмування, а їх підтримка може стати трудомісткою при зміні вимог або структури системи. Також автоматизація не завжди ефективна для оцінки аспектів зручності користування та інтуїтивності інтерфейсу, які потребують безпосередньої взаємодії людини.

Інтелектуальні та сучасні методи тестування програмного забезпечення за ступенем автоматизації

Інтелектуальні та сучасні методи тестування програмного забезпечення за ступенем автоматизації ґрунтуються на підході використанні штучного інтелекту (AI), алгоритмів машинного навчання та аналітичних систем для автоматизації та оптимізації процесів перевірки якості.

Основна мета таких методів тестувань – це підвищення ефективності тестування, скорочення часу на знаходження критичних дефектів та мінімізація впливу людського фактору. Згідно з дослідженнями IEEE у галузі AI for Software Testing, використання машинного навчання дозволяє підвищити ефективність виявлення дефектів та скоротити час тестування за рахунок аналізу великих обсягів даних.

Важливу роль із ключових аспектів інтелектуального тестування – є AI-генерація тест-кейсів, яка дозволяє автоматично створювати сценарії перевірки програмного продукту на основі аналізу вимог, історії змін коду та попередніх

дефектів. Такі тест-кейси адаптуються до зміни функціональності і можуть охоплювати складні або часом виникаючі ситуації, що складно передбачити вручну.

Одним із сучасних методів тестування є **self-healing tests**, які дозволяють автоматично адаптувати тестові сценарії при зміні структури програми або інтерфейсу. Це забезпечує стабільність та безперервність автоматизованого тестування, зменшуючи витрати на підтримку тестів та зменшуючи ризик пропуску дефектів.

Також, метод **predictive defect analysis** передбачає використання алгоритмів машинного навчання для прогнозування можливих дефектів у програмному коді. Завдяки аналізу історії помилок, кодових патернів та змін у проєкті можна заздалегідь визначати проблемні ділянки та ефективніше планувати тестування.

Інтелектуальні методи також застосовуються для аналізу логів та поведінки користувачів. Системи AI можуть автоматично ідентифікувати аномалії у роботі програми, нестандартні шляхи виконання сценаріїв або потенційно небезпечні ситуації, що дає змогу запобігти помилкам у продуктивному середовищі ще до їхнього масового виникнення.

Перевагою інтелектуальних методів є їх адаптивність, масштабованість та висока точність виявлення критичних дефектів. Вони дозволяють оптимізувати процес тестування, зменшити повторювані рутинні операції та забезпечити більш ефективний контроль якості у великих і складних проєктах.

Разом із тим, такі методи мають певні обмеження. Використання AI та машинного навчання потребує великого обсягу даних для навчання моделей, високої кваліфікації фахівців та відповідної інфраструктури. Крім того, складні алгоритми можуть бути менш прозорими, що ускладнює інтерпретацію отриманих результатів і прийняття рішень.

Метрики якості програмного забезпечення

Метрики якості програмного забезпечення представляють собою кількісні показники, що дозволяють оцінити рівень відповідності програмного продукту встановленим вимогам, його надійність, продуктивність та зручність використання. Згідно з підходами ISO (ISO/IEC 25010), оцінка якості базується на вимірюваних характеристиках, що відображають функціональні та нефункціональні властивості системи.

Підхід застосування метрик якості полягає у визначенні стану програмного забезпечення, знаходженні слабких місць і оцінці ризиків експлуатації. Метрики допомагають формалізувати результати тестування, забезпечити порівнянність різних версій продукту та планувати подальші заходи з підвищення якості.

Ключові метрики якості програмного забезпечення та спосіб їх використання для оцінки стабільності, продуктивності та надійності системи відображено у таблиці 1.2.

Таблиця 1.2

Ключові метрики якості програмного забезпечення та їх характеристика

Метрика	**Призначення**	**Переваги**
Code coverage	Частка коду, охопленого тестами	Виявлення неперевірених ділянок
Defect density	Кількість дефектів на 1000 рядків коду	Оцінка стабільності системи
MTBF	Середній час між відмовами	Оцінка надійності
Time to detect defect	Час знаходження помилки після її появи	Контроль оперативності тестування
Test pass rate	Відсоток успішно пройдених тестів	Загальна оцінка якості системи

Стандартизовані моделі, такі як ISO/IEC 25010, використовуються для всебічної оцінки програмного забезпечення, визначаючи, наскільки продукт виконує свої функції, наскільки він надійний і стабільний, ефективний у роботі,

сумісний із іншими системами, безпечний та зручний для користувачів, а також наскільки його легко підтримувати. Використання цих моделей гарантує систематичність і об’єктивність аналізу.

Перевагою метрик якості є кількісна оцінка характеристик програмного забезпечення, що дозволяє робити об’єктивні висновки, планувати процес тестування та оцінювати ефективність застосованих методів перевірки. Метрики дозволяють визначати пріоритетні ділянки коду для тестування та концентрувати ресурси на найбільш критичних компонентах.

Разом із тим, застосування метрик потребує правильної інтерпретації результатів. Наприклад, високий показник охоплення коду не гарантує відсутності дефектів, а низька щільність помилок може бути результатом недостатньої кількості тестів. Тому метрики слід використовувати комплексно, поєднуючи кількісні показники з якісним аналізом.

1.3 Порівняльний аналіз підходів тестування та оцінки якості програмного забезпечення

Після розгляду основних груп підходів тестування та оцінки якості програмного забезпечення стає очевидним, що кожен підхід має свої сильні сторони та обмеження. Це дозволяє визначати, які підходи найбільш доцільно застосовувати для конкретних завдань та типів систем.

Класичні методи тестування – добре підходять для невеликих проєктів та перевірки окремих компонентів. Вони дозволяють детально контролювати виконання кожного тестового сценарію та зрозуміло документувати результати. Проте такі методи значно витратні за часом і не завжди ефективні для великих систем, де багато повторюваних перевірок.

Автоматизовані методи – прискорюють процес тестування та забезпечують високу повторюваність перевірок. Вони особливо корисні для регресійного тестування та перевірки стабільності програмних продуктів при частих оновленнях. Основна складність – це початкові витрати на створення та

підтримку автоматизованих тестів, а також необхідність володіння спеціалізованими інструментами.

Інтелектуальні та сучасні методи – засновані на машинному навчанні та штучному інтелекті, здатні адаптуватися до змін у системі, передбачати дефекти та формувати нові тестові сценарії аналізу попередніх результатів. Вони особливо ефективні для великих та складних систем, проте їхнє впровадження потребує значних даних для навчання моделей і спеціальних навичок у команді.

Для наочності переваг та недоліків різних підходів наведено порівняльну таблицю 1.3:

Таблиця 1.3

Порівняльна характеристика різних методів тестування за ступенем автоматизації

Метод	**Основні сильні сторони**	**Основні обмеження**
Класичні методи	Простота, контроль над кожним тестом, прозорість результатів	Трудомісткість, низька масштабованість, високі витрати часу
Автоматизовані методи	Прискорення тестування, повторюваність, ефективність для великих систем	Витрати на розробку та підтримку тестів, обмежена гнучкість
Інтелектуальні та сучасні методи	Адаптивність, прогнозування дефектів, можливість автоматичного генерування тестів	Потреба у великих даних, складність реалізації, ресурсоємність

Проведений аналіз свідчить про те, що жоден із розглянутих методів не є універсальним рішенням для всіх типів програмних систем. Найбільш ефективною стратегією є комбінування методів:

- Класичні методи забезпечують базову перевірку вимог.
- Автоматизовані – прискорюють регресійне тестування.
- Інтелектуальні та сучасні методи – передбачають дефекти на ранніх етапах.

Вибір оптимальної комбінації визначається складністю системи, вимогами до якості та наявними ресурсами команди розробки.

Моделі якості програмного забезпечення

Моделі якості програмного забезпечення являють собою формалізовані підходи та концептуальні рамки, що визначають основні характеристики програмного продукту та способи їх оцінки. Вони дозволяють систематизувати процес тестування, оцінювання та підтримки програмних систем, забезпечуючи об’єктивність і узгодженість результатів.

Метою застосування моделей якості є визначення ключових параметрів продукту, які впливають на його функціональність, надійність, безпеку та зручність використання. Моделі дозволяють визначити, що слід вимірювати, які метрики використовувати і як інтерпретувати отримані дані для прийняття управлінських рішень.

Основні моделі якості програмного забезпечення та їх ключові характеристики, що забезпечують систематизацію підходів до оцінки програмного забезпечення, наведено у таблиці 1.4:

Таблиця 1.4

Моделі якості програмного забезпечення

Модель якості	**Основні характеристики**	**Переваги використання**
ISO/IEC 25010	Функціональність, надійність, продуктивність, безпека, сумісність, зручність, супроводжуваність, переносимість	Універсальність, системність, стандартизація
McCall’s model	Надійність, ефективність, тестованість, зрозумілість, інтегрованість	Розділення факторів якості, історична значущість
Boehm’s model	Експлуатаційна та розробницька якість, продуктивність, надійність	Врахування життєвого циклу продукту

Моделі якості забезпечують структуру для визначення критеріїв оцінки програмного забезпечення. Вони дозволяють об’єднати кількісні метрики (наприклад, code coverage, defect density) та якісні характеристики (зручність використання, безпека), створюючи комплексну базу стану продукту. Перевагою застосування моделей є їхня універсальність та системність. Вони дозволяють стандартизувати процес оцінки якості для різних типів програмних продуктів і забезпечують зрозумілий підхід до порівняння різних версій або систем. Водночас, моделі якості не завжди враховують специфічні умови конкретного проєкту або галузі. Тому для практичного використання часто застосовується адаптація стандартних моделей під потреби конкретного програмного забезпечення та команди розробки.

1.4 Постановка задачі

Сучасні програмні системи характеризуються зростаючою складністю архітектури, великою кількістю взаємозалежних модулів та динамічним розвитком функціоналу. Це висуває підвищені вимоги до якості програмного забезпечення та процесів його перевірки. Разом із тим, значна частина розробників, особливо на рівні індивідуальних та навчальних проєктів не застосовує систематичного підходу до тестування: відсутні тест-плани, відсутнє покриття коду тестами, а оцінка якості здійснюється виключно суб’єктивно.

Проблема полягає у відсутності практично відпрацьованої методики, яка дозволяла б: обрати відповідні підходи тестування залежно від типу програмного модуля; застосувати їх системно з використанням сучасних інструментів автоматизації; кількісно оцінити якість програмного забезпечення за стандартизованими метриками.

Метою даної роботи є дослідження методів тестування програмного забезпечення та практична застосування підходу до перевірки якості на програмному застосунку з об’єктно-орієнтованою архітектурою.

Для досягнення поставленої мети необхідно вирішити такі задачі:

1. Проаналізувати існуючі методи тестування програмного забезпечення, а саме: «чорної», «білої», «сірої» скриньок

2. Дослідити рівні тестування до кожного виду методу.

3. Розробити тест-план для програмного застосунку, обґрунтувати вибір методів тестування для кожного модуля та встановити критерії якості.

4. Реалізувати функціональні тест-кейси методом «чорної скриньки» для перевірки ігрової механіки застосунку.

5. Реалізувати набір автоматизованих тестів із використанням методів та використанням фреймворку pytest.

6. Реалізувати набір автоматизованих модульних тестів з використанням методів та використанням фреймворку pytest та плагіну для вимірювання покриття коду.

7. Реалізувати систему логування подій на базі стандартної бібліотеки logging для інтеграційного тестування з використанням методів.

8. Виміряти покриття коду та проаналізувати результати виконання тестів, класифікувати виявлені дефекти.

9. Оцінити якість програмного забезпечення відповідно до міжнародного стандарту ISO/IEC 25010 та сформулювати рекомендації щодо її підвищення.

Об'єктом дослідження є прикладне програмне забезпечення «Cosmic Heat» –комп'ютерна гра з об'єктно-орієнтованою архітектурою, розроблена на мові Python з використанням бібліотеки pygame.

Предметом дослідження є методи тестування програмного забезпечення – «чорної скриньки», «білої скриньки» та «сірої скриньки», а також метрики оцінки якості за стандартом ISO/IEC 25010.

Вхідними даними є вихідний програмний код застосунку «Cosmic Heat», розподілений на вісім модулів пакету classes/, та технічні вимоги до його функціональності й продуктивності.

Очікуваними результатами є:

- Розроблені та виконані набори тестів трьома методами: функціональними тест-кейсами, автоматизованих модульних тестів, інтеграційних тест-кейсів.
- Кількісні показники якості: покриття коду не менше 70%,
- Класифікація виявлених дефектів та обґрунтований висновок щодо якості програмного забезпечення за характеристиками ISO/IEC 25010.
- Порівняльний аналіз трьох методів тестування з висновком щодо доцільності їх комплексного застосування.

РОЗДІЛ 2. РІВНІ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОЦІНКИ ЯКОСТІ ЙОГО ФУНКЦІОНУВАННЯ

2.1 Рівні тестування програмного забезпечення

Тестування програмного забезпечення виконується поетапно та охоплює різні рівні перевірки системи. Кожен рівень тестування має власну мету, область перевірки та типи дефектів, які можуть бути виявлені під час тестування.

Поділ тестування на рівні дозволяє систематизувати процес забезпечення якості програмного забезпечення та забезпечити поступову перевірку системи – від окремих програмних компонентів до повністю готового програмного продукту.

У сучасній розробці програмного забезпечення зазвичай виділяють чотири основні рівні тестування:

- Unit Testing (модульне тестування)
- Integration Testing (інтеграційне тестування)
- System Testing (системне тестування)
- Acceptance Testing (приймальне тестування)

Кожен наступний рівень тестування базується на результатах попереднього та дозволяє виявляти помилки різного типу – від дефектів окремих функцій до проблем взаємодії компонентів і невідповідності бізнес-вимогам (Рис. 2.1).

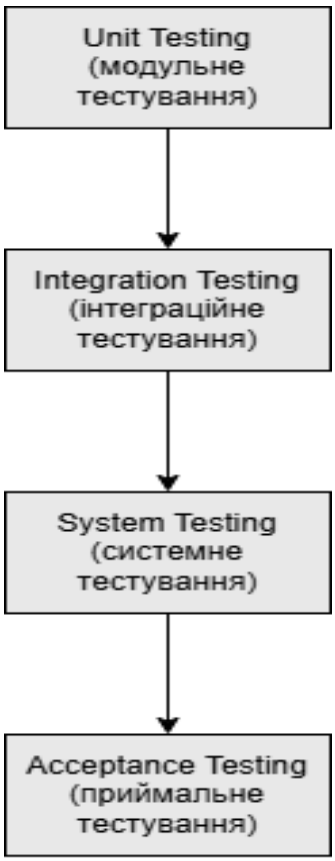


Рис. 2.1 Рівні тестування

2.2 Модульний рівень тестування (Unit testing)

Являє собою перевірку окремих компонентів програмного забезпечення (функцій, методів, класів) із метою виявлення помилок на найнижчому рівні реалізації. Таке тестування виконується ізольовано від інших частин системи, що дозволяє точно локалізувати дефекти.

Основною метою модульного тестування є перевірка правильності роботи окремих елементів програми ще на ранніх етапах розробки. Завдяки цьому помилки можна виявити до інтеграції компонентів між собою, що значно зменшує витрати часу на подальше виправлення дефектів.

Модульні тести зазвичай створюються самими розробниками під час написання коду. Кожен тест перевіряє конкретний сценарій роботи функції або модуля та порівнює отриманий результат із очікуваним.

У таблиці 2.1 представлені основні характеристики модульного тестування:

Таблиця 2.1

Основні характеристики модульного тестування

Характеристика	**Опис**
Рівень тестування	Перший рівень тестування
Об’єкт перевірки	Функція, метод, клас, модуль
Виконується	Розробником
Основна мета	Перевірка коректності окремих компонентів
Швидкість виконання	Висока
Ізоляція	Компоненти тестуються незалежно
Метод за ступенем автоматизації	Переважно автоматизоване тестування

У сучасній розробці програмного забезпечення модульне тестування переважно виконується автоматизовано. Автоматизація дозволяє регулярно виконувати тести після внесення змін у код та швидко виявляти дефекти. Автоматизовані unit-тести інтегруються із системами безперервної інтеграції та доставки програмного забезпечення (CI/CD), що забезпечує постійний контроль якості програмного продукту протягом усього життєвого циклу розробки.

Модульне тестування реалізується за допомогою фреймворків, що надають засоби для опису тестових випадків, виконання перевірок та формування звітів про результати. Серед найбільш поширених інструментів виділяють pytest, JUnit та TestNG.

Pytest у модульному тестуванні має широке застосування зумовлене простотою синтаксису, гнучкою системою налаштування та розвиненою екосистемою плагінів. Фреймворк підтримує параметризацію тестів, що дозволяє виконувати один тестовий сценарій із різними наборами вхідних даних, а також механізм фікстур для підготовки та очищення тестового середовища. Важливою перевагою pytest є детальні звіти про помилки, які суттєво спрощують локалізацію дефектів.

Фікстури (fixtures) є одним із ключових механізмів фреймворку pytest, що забезпечують підготовку та очищення тестового середовища.

Основна ідея фікстур полягає у тому, що вони виконуються автоматично перед тестом (або після нього) і передають необхідні ресурси безпосередньо як аргументи тестової функції. Це дозволяє уникнути дублювання коду підготовки середовища у кожному тестовому сценарії.

Важливою особливістю фікстур є можливість визначення їхньої області видимості через параметр scope, що визначає, як часто фікстура створюється та знищується:

function – фікстура створюється для кожного тесту окремо.

class – одна фікстура на весь клас тестів.

module – одна фікстура на весь файл.

session – одна фікстура на весь виконання тестів.

Механізм фікстур у pytest організований за ієрархічним принципом (Рис.2.2), в основі якого лежить поняття області видимості (scope). Область видимості визначає життєвий цикл фікстури – коли саме вона створюється, скільки часу існує та коли знищується. Діаграма наочно відображає цю ієрархію від найширшого рівня (зверху) до найвужчого (знизу), де test_order є кінцевим тестом, що використовує всі фікстури вище нього.

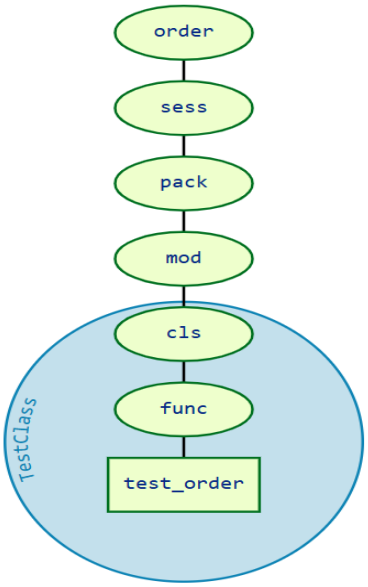


Рис. 2.2 Механізм фікстур

JUnit є стандартним фреймворком для модульного тестування у мові Java та входить до базового інструментарію більшості Java-проєктів. Актуальна версія JUnit6 суттєво розширила можливості попередніх версій: запроваджено підтримку вкладених тестів, умовне виконання тестових сценаріїв, а також розширений механізм анотацій. JUnit інтегрується з провідними середовищами розробки – IntelliJ IDEA та Eclipse – а також із системами збірки Maven та Gradle, що забезпечує його органічне включення у процес CI/CD. Фреймворк підтримує концепцію тестових суїтів, що дозволяє групувати та ініціалізувати набори тестів у визначеній послідовності.

TestNG (Testing Next Generation) є фреймворком для тестування на платформі Java, розробленим як альтернатива JUnit із розширеними можливостями. Ключовою особливістю TestNG є підтримка групування тестів, паралельного виконання та гнучкої конфігурації через XML-файли. Фреймворк надає розвинені засоби для організації тестових залежностей, тобто визначення порядку виконання тестів та умов виконання. Завдяки вбудованій підтримці DataProvider, TestNG ефективно застосовується для реалізації тестів із великою кількістю варіантів вхідних даних. Порівняльну характеристику інструментів модульного тестування наведено у таблиці 2.2:

Таблиця 2.2

Порівняльна характеристика інструментів модульного тестування

Критерій	**pytest**	**JUnit 5**	**TestNG**
Мова програмування	Python	Java	Java
Параметризація	@pytest.mark.parametrize	@ParametrizedTest	@DataProvider
Паралельне виконання	Через плагіни	Обмежена підтримка	Вбудована підтримка
Інтеграція з CI/CD	Легка	Повна	Повна
Групування тестів	Маркери (@pytest.mark)	Теги (@Tag)	Групи(groups)
Звітність	Базова + плагіни	Вбудована	Розширена

Модульне тестування використовується для:

- Перевірки правильності реалізації окремих функцій.
- Виявлення помилок на ранніх етапах розробки.
- Спрощення процесу супроводу програмного забезпечення.
- Забезпечення стабільності коду після внесення змін.
- Полегшення рефакторингу.
- Підвищення загальної якості програмного забезпечення.

Під час модульного тестування кожна функція або модуль перевіряється окремо. Для цього створюються тестові сценарії, які подають вхідні дані та перевіряють, чи відповідає результат очікуваному значенню.

Наприклад, якщо існує функція для обчислення суми двох чисел, тест перевіряє:

- Чи правильно обчислюється результат?
- Чи працює функція з різними типами даних?
- Чи коректно обробляються помилкові значення?

Модульне тестування базується на перевірці окремих програмних компонентів за допомогою спеціально підготовлених тестових сценаріїв. Кожен

тест містить вхідні дані, очікуваний результат та механізм перевірки фактичного результату роботи функції або модуля. Основною ідеєю структури модульного тестування є ізольована перевірка окремої частини програми без впливу зовнішніх компонентів системи. Такий підхід дозволяє швидко локалізувати помилки та спростити процес налагодження програмного забезпечення.

Нижче наведено спрощену структуру процесу модульного тестування (Рис 2.3):

Рис. 2.3 Структура процесу модульного тестування

У представленій схемі показано базову структуру модульного тестування. Спочатку обирається окремий програмний модуль або функція, після чого створюється тестовий сценарій із визначеними вхідними даними та очікуваним

результатом. Під час виконання тесту фактичний результат порівнюється з очікуваним значенням, що дозволяє визначити правильність роботи компонента.

Однією з головних особливостей модульного тестування є ізоляція компонентів. Під час тестування модуль не повинен залежати від:

- Бази даних.
- Мережевих сервісів.
- Зовнішніх API.
- Файлової системи.
- Інших модулів.

Для цього часто використовуються спеціальні об’єкти які представлені у таблиці 2.3:

Таблиця 2.3

Спеціальні об’єкти модульного тестування

Технологія	**Призначення**
Mock	Імітація поведінки об’єкта
Stub	Повернення підготовлених даних
Fake	Спрощена реалізація компонента

Також у модульному тестуванні є свої недоліки, які представлені у таблиці 2.4:

Таблиця 2.4

Недолік	**Опис**
Не перевіряє взаємодію модулів	Для цього потрібне інтеграційне тестування
Потребує часу на написання	Необхідно створювати окремі тестові сценарії
Складність підтримки	При зміні логіки програми тести потрібно оновлювати
Неможливість повної перевірки системи	Перевіряються лише окремі компоненти

2.3 Інтеграційний рівень тестування (Integration Testing)

Рівень тестування програмного забезпечення, при якому перевіряється взаємодія між окремими модулями, компонентами або підсистемами програмного продукту. Після успішного проходження модульного тестування окремі компоненти системи об’єднуються між собою, після чого виконується перевірка правильності їхньої спільної роботи.

Основною метою інтеграційного тестування є виявлення помилок, пов’язаних із взаємодією між модулями, передачею даних, викликом функцій та інтеграцією із зовнішніми сервісами.

Інтеграційне тестування займає проміжне місце між модульним та системним тестуванням і дозволяє переконатися, що окремі частини системи коректно працюють у складі єдиного програмного середовища.

Представлено таблицю 2.5 основних характеристик інтеграційного тестування:

Таблиця 2.5

Основні характеристики інтеграційного тестування

Характеристика	**Опис**
Рівень тестування	Другий рівень тестування
Об’єкт перевірки	Взаємодія модулів та компонентів
Основна мета	Перевірка обміну даними між компонентами
Виконується	Після Unit Testing
Тип помилок	Помилки інтеграції та взаємодії
Метод за ступенем автоматизації	Частково або повністю автоматизований

Інтеграційне тестування спрямоване на перевірку коректності взаємодії між компонентами системи. Отже, інструменти для його реалізації мають забезпечувати можливість тестування між-модульних зв’язків, обмін даними. Як наочно демонструє діаграма Венна (Рис.2.4), інтеграційне тестування зосереджується саме на зоні перетину між окремими модулями системи – тобто на точках їхньої взаємодії, а не на внутрішній логіці кожного модуля окремо.

Саме у цій зоні найчастіше виникають дефекти, пов’язані з некоректною передачею даних, невідповідністю інтерфейсів або порушенням логіки інтеграції. У цьому контексті широко використовуються pytest, TestNG та Postman.

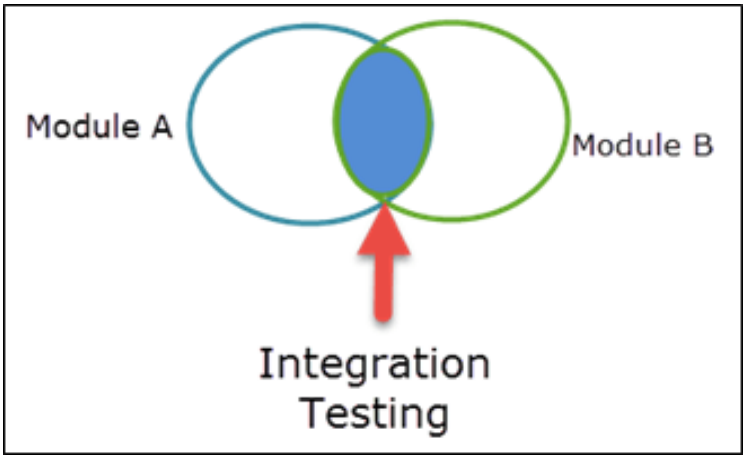


Рис. 2.4 Інтеграційне тестування

Pytest – популярний фреймворк для тестування інтеграційним рівнем. У контексті інтеграційного тестування застосовується для перевірки взаємодії між модулями Python-застосунку до прикладу Pycharm, тестування внутрішніх API та перевірки коректності передачі даних між компонентами.

TestNG при інтеграційному тестуванні використовується для організації складних тестових сценаріїв із залежностями між компонентами.

Фікстури pytest відіграють важливу роль також і в контексті інтеграційного тестування. Якщо при модульному тестуванні фікстури використовуються переважно для ізоляції окремих компонентів, то при інтеграційному – вони забезпечують автоматичне розгортання спільних ресурсів, необхідних для перевірки взаємодії між модулями.

Зокрема, за допомогою фікстур із scope="session" або scope="module" можна один раз ініціалізувати тестову базу даних, запустити тестовий HTTP-сервер або створити клієнт для роботи з API, після чого цей ресурс буде доступний для всіх інтеграційних тестів у межах визначеної області видимості. Після завершення тестів фікстура через механізм yield автоматично виконує

очищення середовища, що гарантує ізольованість кожного тестового сценарію та відтворюваність результатів незалежно від порядку їх ініціалізації.

У проєктах інтеграційного тестування, TestNG часто використовується для перевірки:

- Взаємодії між backend-сервісами.
- Роботи REST API.
- Інтеграції з базами даних.
- Обміну даними між модулями системи.

Можливість визначення залежностей між тестовими методами через анотацію @DependsOnMethods дозволяє моделювати реальні сценарії взаємодії модулів. Паралельне виконання тестів TestNG скорочує загальний час перевірки при великій кількості інтеграційних сценаріїв.

Postman – є одним із найбільш поширених інструментів для тестування REST API та інтеграційного тестування на рівні програмних інтерфейсів.

Інструмент має зручний графічний інтерфейс для формування HTTP-запитів усіх типів:

- GET – використовується для отримання даних з серверу.
- POST – застосовується для нових ресурсів на сервері.
- PUT –призначений для повного оновлення існуючого ресурсу.
- DELETE – використовується для видалення ресурсу з серверу.
- PATCH – застосовується для часткового оновлення ресурсу

Також аналізу відповідей сервера та написання автоматизованих перевірок на мові JavaScript. Ключовою можливістю Postman є організація запитів у колекції та створення середовищ (environments) із змінними, що забезпечує гнучке управління конфігурацією тестів.

Ще, крім підтримки різних типів запитів, Postman надає такі можливості:

- Формування та редагування HTTP-заголовків і параметрів запиту
- Робота з різними форматами даних (JSON, XML, form-data)

- Автоматична перевірка відповідей сервера (status code, тіло відповіді)
- Створення колекцій запитів для повторного використання
- Автоматизація тестів за допомогою вбудованих скриптів
- Інтеграція з CI/CD для виконання тестів у конвеєрі розробки

Інтеграційне тестування використовується для:

- Перевірки взаємодії між модулями системи.
- Перевірки правильності передачі даних.
- Виявлення помилок у взаємодії компонентів.
- Перевірки інтеграції з базами даних.
- Тестування взаємодії із зовнішніми API та сервісами.
- Перевірки роботи підсистем після об’єднання.

На відміну від модульного тестування, де компоненти перевіряються ізольовано, інтеграційне тестування оцінює спільну роботу декількох модулів одночасно.

Наприклад:

- Модуль авторизації взаємодіє з базою даних.
- Веб-інтерфейс передає запити серверній частині.
- API взаємодіє із зовнішнім сервісом.
- Система обробки платежів працює разом із банківським шлюзом.

У процесі тестування перевіряється, чи правильно передаються параметри між компонентами, чи коректно обробляються відповіді та чи не виникають помилки під час взаємодії.

Також інтеграційне тестування має свої недоліки які представлено у таблиці 2.6:

Таблиця 2.6

Недоліки інтеграційного тестування

Недолік	**Опис**
Складність налаштування	Потрібне тестове середовище
Залежність від інших модулів	Тестування неможливе без інтеграції
Повільніше виконання	Порівняно з unit-тестами
Важча локалізація дефектів	Проблема може виникати між кількома модулями

2.4 Системний рівень тестування (System Testing)

Являє собою рівень тестування програмного забезпечення, при якому перевіряється робота всієї системи в цілому відповідно до функціональних та нефункціональних вимог.

На цьому етапі тестування програмний продукт розглядається як завершена та інтегрована система. Перевіряється не окремий модуль або взаємодія компонентів, а повна працездатність усієї програми в умовах, максимально наближених до реального використання.

На відміну від попередніх етапів, які перевіряють окремі компоненти або їх взаємодію (Рис.2.5), системне тестування розглядає програмний продукт як єдине ціле – у середовищі, максимально наближеному до реального. Основною метою є перевірка відповідності системи встановленим функціональним та нефункціональним вимогам, а також коректності сценаріїв взаємодії користувача з інтерфейсом.

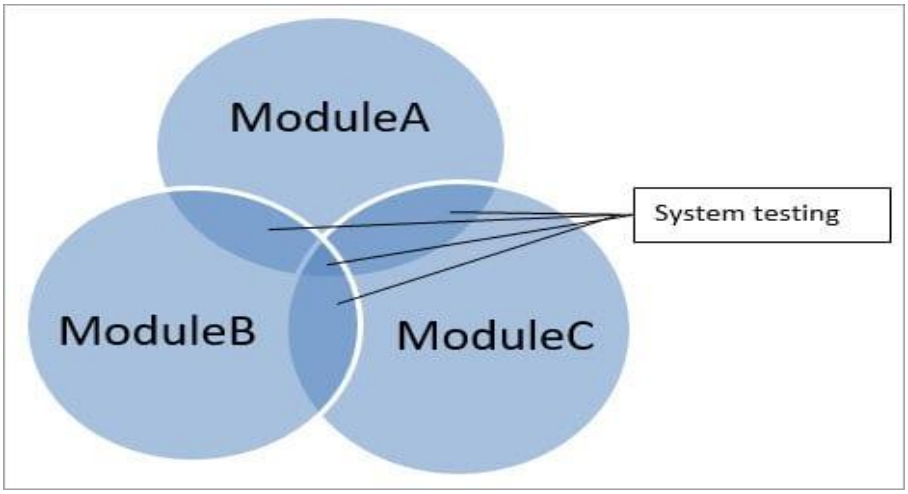


Рис. 2.5 Системне тестування

Центральне місце у системному тестуванні займає перевірка графічного інтерфейсу користувача (UI), оскільки саме через нього кінцевий користувач взаємодіє з системою.

Представлено основні характеристики системного тестування у таблиці 2.7:

Таблиця 2.7

Основні характеристики системного тестування

Характеристика	**Опис**
Рівень тестування	Третій рівень тестування
Об’єкт перевірки	Повна система
Основна мета	Перевірка функціональності всієї системи
Виконується	Після Integration Testing
Тип тестування	Функціональне та нефункціональне
Середовище	Наближене до реального

Однією з головних особливостей системного тестування є комплексний підхід до перевірки програмного забезпечення. У процесі тестування оцінюється робота всієї системи як єдиного програмного продукту.

На цьому етапі перевіряються:

- Взаємодія всіх компонентів системи.

- Коректність бізнес-логіки.
- Стабільність роботи програмного забезпечення.
- Робота інтерфейсу користувача.
- Продуктивність системи.
- Безпека обробки даних.
- Сумісність із різними платформами та пристроями.

Особлива увага приділяється реальним сценаріям використання системи, що дозволяє оцінити готовність програмного продукту до експлуатації.

Під час тестування оцінюються:

- Функціональні можливості.
- Інтерфейс користувача.
- Взаємодія компонентів.
- Продуктивність.
- Безпека.
- Стабільність роботи.
- Сумісність із різними платформами.
- Обробка помилок.

Системне тестування зазвичай виконується спеціалістами з тестування (QA Engineers) у тестовому середовищі, максимально наближеному до робочого.

Системне тестування охоплює як функціональні, так і нефункціональні перевірки програмного забезпечення.

Функціональне тестування спрямоване на перевірку правильності реалізації функцій системи відповідно до технічних вимог та бізнес-логіки. На цьому етапі перевіряються сценарії роботи користувача, обробка даних та правильність виконання операцій.

Нефункціональне тестування оцінює характеристики системи, які не пов’язані безпосередньо з функціональністю, але впливають на якість програмного забезпечення. До таких характеристик належать продуктивність, безпека, стабільність, масштабованість та зручність використання.

Поєднання функціонального та нефункціонального тестування дозволяє комплексно оцінити якість програмного продукту.

Системне тестування є одним із найважливіших етапів забезпечення якості програмного забезпечення. Саме на цьому рівні перевіряється готовність програмного продукту до використання в реальних умовах. Дозволяє виявити помилки, які неможливо знайти під час модульного або інтеграційного тестування, оскільки вони проявляються лише при повноцінній роботі всієї системи. Результати системного тестування суттєво впливають на стабільність, надійність та якість програмного забезпечення перед передачею продукту кінцевим користувачам.

Також системне тестування має свої недоліки, які представлені у таблиці 2.8:

Таблиця 2.8

Недоліки системного тестування

Недолік	**Опис**
Висока складність	Велика кількість сценаріїв
Тривале виконання	Тестування всієї системи
Високі витрати ресурсів	Необхідне окреме середовище
Складність автоматизації	Не всі сценарії легко автоматизувати

2.5 Приймальний рівень тестування (Acceptance Testing)

Завершальний рівень тестування програмного забезпечення, метою якого є перевірка готовності системи до використання кінцевими користувачами або замовником.

На цьому етапі оцінюється, наскільки програмний продукт відповідає бізнес-вимогам, технічному завданню та очікуванням користувачів. Acceptance

Testing виконується після успішного завершення системного тестування та є фінальною перевіркою перед введенням програмного забезпечення в експлуатацію.

Основне завдання приймального тестування – підтвердити, що система готова до реального використання та може бути передана замовнику або випущена у production-середовище.

Однією з основних особливостей приймального рівня тестування – є активна участь замовника або кінцевих користувачів у процесі перевірки системи. Саме користувачі оцінюють, наскільки програмний продукт відповідає реальним потребам та бізнес-процесам.

Під час приймального тестування система перевіряється у середовищі, максимально наближеному до production-середовища. Це дозволяє оцінити поведінку програмного забезпечення в умовах, наближених до реальної експлуатації.

Представлено таблицю 2.9 основних характеристик приймального тестування:

Таблиця 2.9

Основні характеристики приймального тестування

Характеристика	**Опис**
Рівень тестування	Четвертий (фінальний) рівень
Об’єкт перевірки	Повна система
Основна мета	Перевірка відповідності бізнес-вимогам
Виконується	Після System Testing
Учасники	Замовники, користувачі, QA
Результат	Підтвердження готовності продукту

Особлива увага приділяється:

- Виконанню бізнес-сценаріїв.
- Правильності роботи функціоналу.
- Стабільності системи.
- Зручності використання.
- Відповідності технічному завданню.
- Готовності продукту до виконання.

Приймальне тестування є фінальним етапом перевірки перед передачею програмного продукту замовнику або публічним релізом системи. Воно орієнтоване на внутрішню реалізацію системи, а на перевірку того, чи задовольняє продукт потреби користувачів та бізнесу.

На цьому етапі особлива увага приділяється:

- Реальним сценаріям використання.
- Зручності роботи користувача.
- Відповідності функціоналу вимогам.
- Стабільності роботи системи.
- Готовності до production-середовища.

Часто виконується в середовищі, максимально наближеному до реального робочого середовища.

Процес приймального тестування поділяється на основні етапи які представлено у таблиці 2.10:

Таблиця 2.10

Основні етапи приймального тестування

Етап	**Опис**
Аналіз вимог	Вивчення бізнес-вимог
Підготовка сценаріїв	Формування реальних тестових сценаріїв
Підготовка середовища	Налаштування production-like середовища
Виконання тестування	Перевірка системи користувачами

Продовження таблиці 2.10

Етап	Опис	Етап
Аналіз результатів	Оцінка виявлених дефектів	Аналіз результатів
Ухвалення рішення	Підтвердження готовності продукту	Ухвалення рішення

Приймальне тестування є фінальним етапом перевірки програмного забезпечення перед його введенням в експлуатацію. Саме Acceptance Testing дозволяє визначити, чи відповідає продукт вимогам замовника та чи готовий він до реального використання.

Успішне проходження приймального тестування свідчить про готовність програмного забезпечення до релізу та передачі кінцевим користувачам.

Також приймальне тестування має свої недоліки які представлено у таблиці 2.11:

Таблиця 2.11

Недоліки приймального тестування

Недолік	Опис
Висока тривалість	Велика кількість сценаріїв
Залежність від користувачів	Необхідна участь замовника
Складність організації	Потрібне production-like середовище
Пізнє виявлення дефектів	Помилки можуть бути дорогими у виправленні

РОЗДІЛ 3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Опис об’єкта тестування

Об’єктом дослідження та практичного тестування у даній роботі є прикладне програмне забезпечення, а саме двовимірна аркадна гра жанру "shoot ‘em up" під назвою «Cosmic Heat» **(**автор –Dave-YP**)**. Реалізована мовою програмування Python з використанням бібліотеки pygame (Рис.3.1).

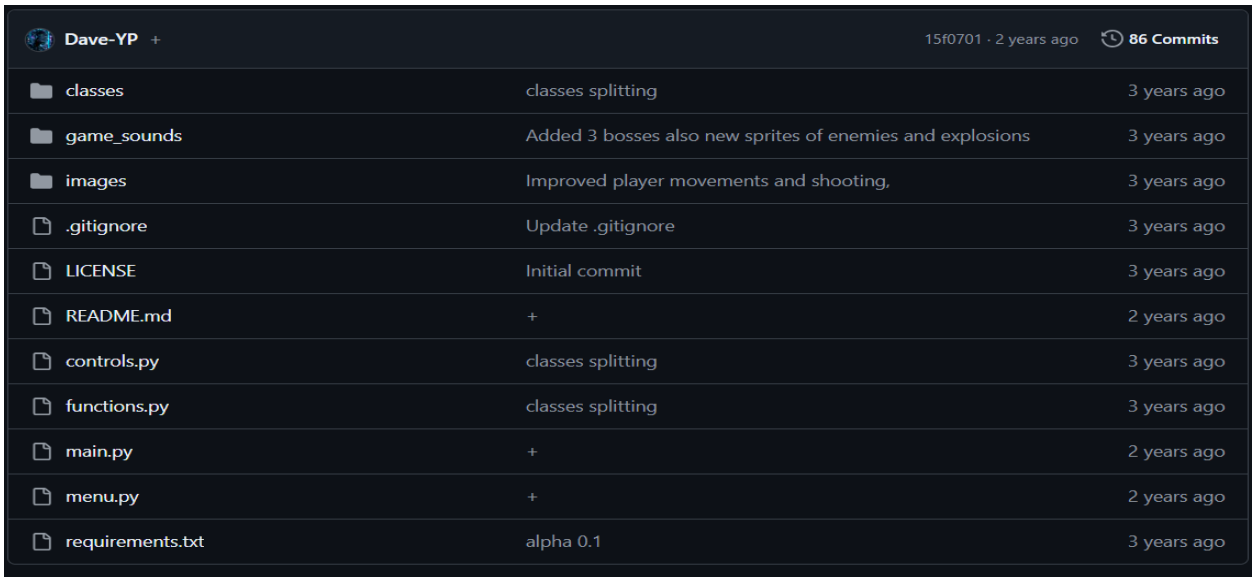


Рис.3.1 Програмне забезпечення (2-D гра) від розробника Dave-YP

Обрана програмне забезпечення містить типові елементи такі як:

- Обробку подій.
- Логіку руху об’єктів.
- Систему зіткнень.
- Механіку нарахування очок.
- Взаємодію між ігровими сутностями та зміну станів під час виконання програми.

За ігровою механікою гравець керує космічним кораблем, який протистоїть хвилям ворожих об’єктів – ворожим кораблям, метеоритам і трьом різним . Керування здійснюється за допомогою клавіш:

- Shoot – клавіша пробіл (K_SPACE).

- Move – клавіші зі стрілками: верх, вліво, вниз, вправо (K_UP; K_LEFT; K_DOWN; K_RIGHT)
- Pause – клавіша P.
- Exit (Вихід) – клавіша Esc.
- Також керування підтримує керування за допомогою джойстику (move_player_with_joystick).

Гравець має два ресурси: здоров’я (player_life) та набої (bullet_counter), обидва обмежені максимальним значенням 200 (Рис.3.2).

Рис.3.2 Ресурси здоров’я та набоїв гравця

Ці ресурси можна поповнювати підбираючи ресурсні об’єкти які з’являються під час гри.

Складність наростає динамічно: з ростом рахунку з’являються нові типи ворогів, збільшується швидкість об’єктів, змінюється поведінка противників і візуальне середовище гри.

- Рівні складності побудована з допомогою системи рахунку, а саме:
- При досягненні 3000 очок починають з’являтися вороги другого типу (Enemy2) і змінюється фонове зображення (background1).
- При досягненні 5000 очок з’являється перший ключовий опонент (Boss1).
- При 10000 очок з’являється другий ключовий опонент (Boss2), та зміна фонового зображення (background3).
- При досягненні 15000 очок з’являється третій ключовий опонент (Boss3) та змінюється фонове зображення (background4).

- Також при досягненні певної кількості очок, паралельно з ростом рахунку збільшується і швидкість метеоритів – від базових 2 пікселів за кадр до максимальних 8 пікселів за кадр.

Така багаторівнева система прогресії робить ігрову логіку нетривіальною і цікавою з точки зору тестування програмного забезпечення – кожен поріг є потенційним місцем для дефекту.

Якщо проаналізувати програмне забезпечення, то архітектура побудована за об’єктно-орієнтованим програмуванням. Всі ігрові об’єкти є нащадками класу pygame.sprite.Sprite, що забезпечує стандартизований інтерфейс, оновлення стану та виявлення колізій. Взаємодія між об’єктами відбувається через групи спрайтів (pygame.sprite.Group) та перевірку колізій за допомогою методу colliderect().

Використання саме ігрового застосунку як об’єкта тестування дозволяє дослідити різні підходи до перевірки якості програмного забезпечення на практиці. У структурі проєкту (Рис.3.3) присутні окремі модулі для керування гравцем, противниками, снарядами, анімаціями вибухів, та іншими елементами гри, що створює зручні умови для проведення тестування.

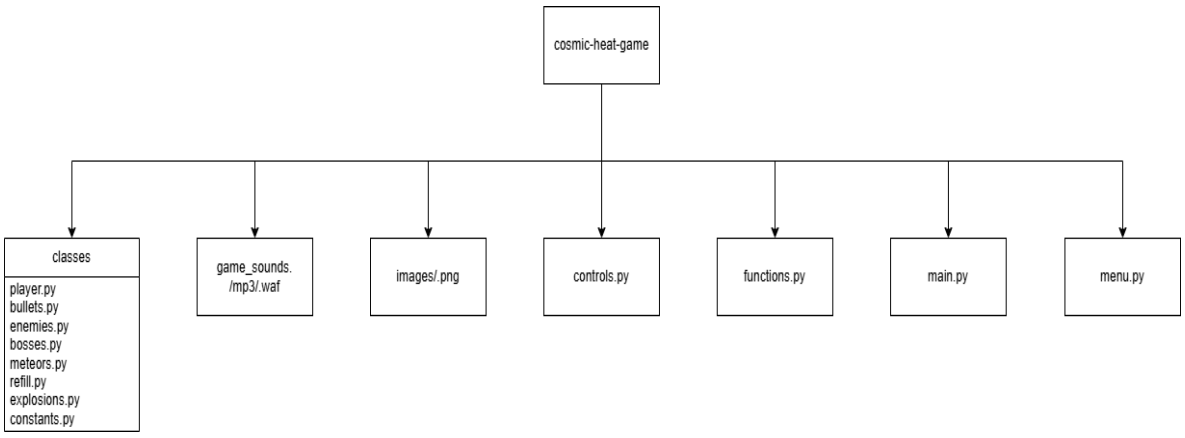


Рис. 3.3 Структура проєкту

Програмне забезпечення складається з восьми модулів у пакеті classes/, кожен з яких реалізує окремий ігровий компонент. Загальна характеристика модулів наведена в таблиці 3.1:

Таблиця 3.1

Модулі програмного забезпечення

Модуль	**Класи**	**Призначення**
player.py	Player	Керований гравцем корабель, логіка руху і меж
bullets.py	Bullet	Снаряди гравця, рух і видалення
enemies.py	Enemy1, Enemy2, Enemy2Bullet	Ворожі кораблі з різною поведінкою
bosses.py	Boss1, Boss2, Boss3 та їхні снаряди	Фінальні ключові опоненти з унікальними механіками
meteors.py	Meteors, Meteors2, BlackHole	Небезпечні об’єкти середовища
refill.py	BulletRefill, HealthRefill, DoubleRefill, ExtraScore	Ресурсні об’єкти для збору
explosions.py	Explosion, Explosion2	Анімація вибухів
constants.py	-	Глобальні константи гри

Вибір даного програмного забезпечення, як об’єкта тестування обумовлений низкою факторів:

По-перше. Достатній обсяг коду для вимірювання покриття, відкритий доступ до вихідного коду.

По-друге. Наявність модульної структури коду дозволяє тестувати кожен клас незалежно від інших, що відповідає принципам модульного тестування.

По-третє. Відсутність існуючих тестів та тестового покриття у репозиторії надає можливість розробити повний набір тестів.

3.2 Розробка тест-плану

Тест-план є основним документом який визначає стратегію обсяг і підходи до тестування програмного забезпечення. В рамках даної роботи тест-план

розроблено відповідно до тестування програмного забезпечення на прикладі прикладного програмного застосунку «Cosmic Heat».

Метою тестування є практична реалізація та порівняння трьох основних методів тестування програмного забезпечення а конкретно:

- **Методу «чорної скриньки» (Black-box testing)**
- **Методу «білої скриньки» (White box testing)**
- **Методу «сірої скриньки» (Grey box testing)**

З подальшою оцінкою якості застосунку на основі отриманих результатів.

До обсягу тестування включено всі функціональні компоненти прикладного програмного забезпечення «Cosmic Heat»:

- Ігровий процес
- Головне меню
- Система керування
- Ігрова механіка
- Продуктивність

В роботі застосовано три методи тестування які відрізняються рівнем доступу до вихідного коду застосунку. Порівняльна характеристика методів наведена в таблиці 3.2:

Таблиця 3.2

Порівняльна характеристика методів тестування

Характеристика	**Чорна скринька**	**Біла скринька**	**Сіра скринька**
Доступ до коду	Відсутній	Повний	Частковий
Що перевіряється	Зовнішня поведінка та програмне забезпечення у цілому	Внутрішня логіка	Продуктивність та запис подій

Продовження таблиці 3.2

Характеристика	**Чорна скринька**	**Біла скринька**
Інструменти	Ручне тестування	pytest, coverage
Результат	Протокол тест-кейсів	Coverage разом з тестами

Критерії якості

Тестування проводиться в логічній послідовності від зовнішнього до внутрішнього рівня.

Спочатку застосовується **метод «чорної скриньки» –**використовується функціональне тестування та тестування зручності використання. Взаємодія з грою, як звичайний гравець – перевіряє керування, ігрову механіку, меню і реакцію на дії гравця. Знання вихідного коду не використовується. Результат фіксується у протоколі тест-кейсів. Без знання коду та доступу до внутрішньої частини (коду).

Потім методом **методом «білої скриньки»** реалізується за дпомогою модульного тестування за допомогою фреймворку pytest. Тестувальник має повний доступ до вихідного коду і розробляє тести що перевіряють внутрішню логіку кожного класу. Для вимірювання повноти тестування використовується інструмент coverage.

Завершується тестування **методом «сірої скриньки» –** реалізується через інтеграційне тестування (поєднанням двох «скриньок») за допомогою вбудування технічного оверлею FPS і системи логування. Гра в гру, як гравець, але паралельно спостерігати поточні технічні дані а саме – середній FPS та система logging для технічних подій.

Така послідовність дозволяє спочатку переконатись що програмне забезпечення функціонує коректно з точки зору гравця а потім дослідити якість внутрішньої реалізації і продуктивність (Табл.3.3):

Таблиця 3.3

Реалізація по етапам

Етап	**Метод**	**Метод за ступенем автоматизації**	**Результат**
1	Чорна скринька	Ручне тестування	Протокол тест-кейсів та функціонал у цілому
2	Біла скринька	Автоматизоване тестування	Набір тестів та coverage
3	Сіра скринька	Комбіноване тестування	FPS, logging

3.3 Тестування методом «чорної скриньки»

Метод чорної скриньки передбачає тестування програмного забезпечення без знання його внутрішньої структури та вихідного коду. Взаємодія з програмою виключно через зовнішній інтерфейс – спостерігає поведінку, реакцію на введення і відповідність очікуваному результату. Було застосовано функціональне тестування – перевірка коректності роботи програмного забезпечення загалом та тестуванням зручності використання.

Середовище тестування

Тестування проводилось на комп’ютері з наступними характеристиками (Табл. 3.4):

Таблиця 3.4

Характеристики середовища тестування

Параметр	**Значення**
Операційна система	Windows 10
Процесор	Intel core i5-9300
Оперативна пам’ять	16 гб
Відеокарта	Gtx 1650
Роздільна здатність	1200 x 800

Функціональне тестування

Функціональне тестування проводилось за заздалегідь складеним протоколом тест-кейсів. Кожен тест-кейс описує конкретну дію гравця, очікуваний результат і фактичний результат спостережений під час гри. Протокол тестування наведено в таблиці 3.5:

Таблиця 3.5

Протокол функціонального тестування

№	Тест-кейс	Кроки виконання	Очікуваний результат	Фактичний результат	Статус
1	Відкрити програмне забезпечення	Відкриття програмного забезпечення	Відкривається головне меню	Меню відкрилось	Pass
2	Навігація меню клавішами	Натиснути клавіши «вверх» та «вниз»	Наведення на клавіші переміщується	Переміщується коректно	Pass
3	Початок гри через меню за допомогою «P lay»	Натиснути клавішу «Enter» на «Play»	Починається ігровий процес	Гра почалась	Pass
4	Рух гравця вліво	Утримувати клавішу «вліво»	Корабель рухається вліво	Рухається	Pass
5	Рух гравця вправо	Утримувати клавішу «вправо»	Корабель рухається вправо	Рухається	Pass
6	Рух гравця вгору	Утримувати клавішу «вверх»	Корабель рухається вгору	Рухається	Pass
7	Рух гравця вниз	Утримувати клавішу «вниз»	Корабель рухається вниз	Рухається	Pass
8	Діагональний рух	Утримувати клавішу вліво разом з клавішею «вверх»	Не буде помилок	Рухається по діагоналі вліво вверх	Pass

Продовження таблиці 3.5

№	Тест-кейс	Кроки виконання	Очікуваний результат	Фактичний результат	Статус
9	Обмеження лівої межі	Утримувати клавішу «вліво» до стіни	Зупиняється на лівій межі	Зупиняється коректно	Pass
10	Обмеження правої межі	Утримувати клавішу «вправо» до стіни	Зупиняється на правій межі	Зупиняється коректно	Pass
11	Обмеження верхньої межі	Утримувати клавішу «вверх» до стіни	Зупиняється на верхній межі	Зупиняється коректно	Pass
12	Обмеження нижньої межі	Утримувати клавішу «вверх» до стіни	Зупиняється на нижній межі	Зупиняється коректно	Pass
13	Стрільба при наявності патронів	Натиснути клавішу «Space»	З’являється постріл і летить прямо	Постріл з’явився і полетів	Pass
14	Стрільба при руху по стронам	Натиснути клавіши «вверх», «вправо» та «Space»	Постріли не змінюють траєкторії і летять прямо	Снаряди летять прямолінійно	Pass
15	Стрільба при 0 патронів	Витратити всі патрони	Корабель нічим не стріляє	Пострілів снарядами не відбувається	Pass
16	Поповнення патронів	Підібрати BulletRefill	Лічильник снарядів збільшився	Снаряди збільшились	Pass
17	Поповнення здоров’я	Підібрати HealthRefill	Лічильник здоров’я збільшився	Здоров’я збільшилось	Pass
18	Пошкодження від метеорита	Зіткнутись з метеоритом	Здоров’я зменшилось	Здоров’я зменшилось	Pass
19	Поява ворогів Enemy1	Почати гру	Вороги рухаються по екрану	З’явились і рухаються	Pass

Продовження таблиці 3.5

№	Тест-кейс	Кроки виконання	Очікуваний результат	Фактичний результат	Статус
21	Стрільба Enemy2	Зачекати поки Enemy2 стріляє	Снаряд летить у напрямку гравця	Снаряд летить вниз до гравця	Pass
22	Поява Boss1	Набрати 5000 очок	Boss1 з’являється зверху	Boss1 з’явився	Pass
23	Стрільба Boss1	Спостерігати за Boss1	Стріляє 3 снарядами одночасно	Стріляє 3 снарядами	Pass
24	Поява Boss2	Набрати 10000 очок	Boss2 з’являється зверху	Boss2 з’явився	Pass
25	Поява Boss3	Набрати 15000 очок	Boss3 з’являється зверху	Boss3 з’явився	Pass
26	Анімація вибуху	Знищити ворога	З’являється анімація вибуху	Анімація відтворилась	Pass
27	Зміна фону	Набрати 3000 очок	Фон змінюється на новий	Фон змінився	Pass
28	Звуки пострілів	Натиснути на «Space»	Звук пострілів	Звук чутно	Pass
29	Звук у меню	Звук у меню, початку гри	Звук музики у меню	Звук чутно	Pass
30	Звук у битві	Звук у процесі гри	Звук та заміна на іншу музику	Музика змінюється та	Pass
31	Game over	Втратити все здоров’я	Показує екран завершення	Екран з’явився	Pass
32	Відображення рахунку	Влетіти і стріляти у ворогів	Рахунок збільшується	Збільшується коректно	Pass
33	Рекорд зберігається	Встановити рекорд	Hi-score залишається	Залишається, не змінюється	Pass
34	Вийти за допомогою клавіші	Натиснути клавішу «Enter» на «Exit»	Вихід з гри	Вихід з програмного забезпечення	Pass

Зручність використання

Зручність використання програмного застосунку інтуїтивно зрозумілий. Було виявлено дефект роботи шкал зодоров'я та набоїв. Обидві шкали на початку гри не знижували свій запас (набоїв чи здоров'я).

Також було б правильним рішенням додати клавішу «вихід у меню» до прикладу де вікно «паузи», щоб можна було вийти під час ігрового сеансу за потреби. Попри це, користуватися програмним забезпеченням цілком можливо.

3.4 Тестування методом «білої скриньки»

Метод білої скриньки (White-Box Testing) базується на повному доступі до вихідного коду програми. При використанні цього методу відомо:

- Внутрішню структуру застосунку які класи існують.
- Які методи вони містять.
- Яка логіка реалізована всередині кожного методу.

Це дозволяє розробляти тести що перевіряють не лише зовнішню поведінку а й конкретні рядки коду, умовні розгалуження і граничні стани.

В рамках даної роботи для методу «білої скриньки» реалізовано за допомогою модульного тестування. Для кожного з 8 модулів пакету classes/ розроблено окремий тестовий модуль що перевіряє внутрішню логіку відповідного класу. Загальна структура розробленого тестового набору наведена в таблиці 3.6:

Таблиця 3.6

Структура тестового набору методу білої скриньки

Тестовий файл	**Модуль що тестується**	**Що перевіряється**	**Кількість тестів**
test_constants.py	constants.py	Коректність значень констант, взаємозв'язки між ними	21
test_player.py	player.py	Логіка руху, обмеження меж, зміна стану direction	52

Продовження таблиці 3.6

Тестовий файл	**Модуль що тестується**	**Що перевіряється**	**Кількість тестів**
test_bullets.py	bullets.py	Рух снаряду, умова видалення, колізії	32
test_enemies.py	enemies.py	Відбивання від стін, таймер стрільби, режим переслідування	50
test_meteors_refill.py	meteors.py, refill.py	Пороги видалення, обмеження координат, рух об’єктів	66
test_bosses.py	bosses.py	Режими поведінки Boss, стрільба, телепортація Boss3	78
test_explosions.py	explosions.py	Анімаційні стани, відтворення звуку, видалення спрайту	44
test_game_logic.py	main.py	Здоров’я, патрони, рахунок, поява ключових опонентів, зміна фону	114
Разом			457

Тестовий набір охоплює чотири категорії перевірок які відповідають різним аспектам внутрішньої логіки застосунку: базова логіка та юніт-тести, математичні моделі, тестування станів та управління ресурсами.

Базова логіка та модульне тестування

Модульне тестування охоплює перевірку найпростіших операцій кожного класу – ініціалізацію об’єктів, коректність початкових значень атрибутів і базові методи. Це фундаментальний рівень тестування білої скриньки перед тим, як перевіряти складну логіку треба впевнитись що об’єкт коректно створюється і має правильні початкові значення.

Наприклад для класу Boss1 перевіряється що при створенні швидкість дорівнює 6, таймер пострілу дорівнює 0 а лічильник пострілів дорівнює 0. Якби будь-яке з цих значень було некоректним, Boss вів би себе неправильно з початку гри. Аналогічно для класу Explosion перевіряється що анімація починається з нульового кадру і звук ще не відтворювався. У таблиці 3.7 предаставлено початкові значення атрибутів:

Таблиця 3.7

Початкові значення атрибутів ключових класів

Клас	**Атрибут**	**Очікуване початкове значення**	**Результат перевірки**
Player	speed	10	Pass
Player	direction	down	Pass
Boss1	speed	6	Pass
Boss1	shoot_timer	0	Pass
Boss1	shots_fired	0	Pass
Boss2	speed	5	Pass
Boss3	teleport_timer	0	Pass
Boss3	teleport_interval	160	Pass
Enemy2	shots_fired	0	Pass
Explosion	frame	0	Pass
Explosion	sound_played	False	Pass
Bullet	speed	10	Pass
Meteors	speed	2	Pass

Математичні моделі та фізика

Ця категорія охоплює перевірку математичних розрахунків закладених в ігрову механіку. Знаючи вихідний код можна точно перевірити не лише напрямок руху а й точне числове значення зміщення за один кадр.

Рух гравця реалізований як пряме додавання швидкості до координати. При speed=10 і натисканні стрілки вліво координата X зменшується на 10 пікселів за кадр. В тесті перевіряється точне числове значення зміщення після виклику move_left() нова координата X має дорівнювати старій мінус 10.

Якби розробник випадково написав speed * 2 замість speed гравець рухався б вдвічі швидше і тест одразу виявив би цей дефект оскільки зміщення склало б 20 пікселів замість очікуваних 10.Рух метеоритів реалізований з обертанням через формулу (angle – 1) % 360. Перевіряється граничний випадок при куті 0 після зменшення має бути 359, а не від'ємне значення. Це математична перевірка, коли оператор % повертає невід'ємний результат але якби було використано просте angle -= 1 без % 360 кут міг би стати від'ємним. Рух Boss2 і Boss3 по діагоналі реалізований з нормалізацією швидкості. При русі по діагоналі (напрямок 45°) швидкість ділиться на $\sqrt{2} \approx 1.414$ щоб загальна швидкість залишалась рівною 5, а не $5\sqrt{2}$.

Різниця координат між Boss і Player ділиться на евклідову відстань що дає одиничний вектор напрямку. Множення на speed дає правильне переміщення. Перевіряється що після кожного кадру відстань між Boss і Player зменшується. Представлено таблицю перевірок математичних моделей руху:

Таблиця 3.8

Математичні моделі руху об'єктів

Об'єкт	**Формула руху**	**Що перевіряється**	**Результат**
Player	rect.x -= speed	Зміщення рівно speed пікселів	Pass
Meteors	angle = (angle-1) % 360	Кут 0 → стає 359 (не -1)	Pass
Boss1	rect.x += sin(t*0.01)*3	Синусоїдальний рух по обох осях	Pass
Boss2/3	speed = 5/√2 ≈ 3.54	Нормалізована швидкість по діагоналі	Pass
Bullet	rect.move_ip(0, -speed)	Рух строго вертикально вгору	Pass

Тестування станів

Знання вихідного коду дозволяє точно визначити всі можливі стани об’єктів і умови переходу між ними. Це неможливо при тестуванні чорною скринькою де стани доводиться вгадувати за зовнішньою поведінкою.

В застосунку Cosmic Heat виявлено три класи з чітко вираженими станами. Клас Enemy2 має два стани – стрільба і переслідування з переходом при shots_fired = 10. Клас Explosion має три стани – очікування, анімація і завершення. Клас Boss3 має незалежний цикл телепортації паралельно з основними станами.

Для кожного стану і кожного переходу розроблено окремий тест. Особлива увага приділялась граничним точкам переходу – значенням безпосередньо до і після умови зміни стану. Наприклад для Enemy2 перевіряються значення shots_fired = 9 (ще стрільба), shots_fired = 10 (гранична точка, вже переслідування) і shots_fired = 11 (впевнено переслідування). Загальна характеристика станів об’єктів та тестування переходів представлена у таблиці 3.9:

Таблиця 3.9

Стани об’єктів і результати тестування переходів

Клас	**Стан 1**	**Стан 2**	**Умова переходу**	**Гранична точка**	**Результат**
Enemy2	Стрільба (speed=4)	Поточний стан (speed=10)	shots_fired ≥ 10	shots_fired = 10	Pass
Boss1	Стрільба	Поточний стан (speed=10)	shots_fired ≥ 20	shots_fired = 20	Pass
Boss2	Стрільба	Поточний стан	shots_fired ≥ 20	shots_fired = 20	Pass
Boss3	Стрільба + телепорт	Поточний стан + телепорт	shots_fired ≥ 20	shots_fired = 20	Pass

Продовження таблиці 3.9

Клас	Стан 1	Стан 2	Умова переходу	Гранична точка	Результат
Explosion	Очікування	Анімація	now - last_update > 60 мс	60 мс	Pass
Explosion	Анімація	Завершення	frame == len(images)	frame = 8	Pass
Player	Будь-який напрямок	Заблокований	rect.left == 0	left = 0	Pass

3.5 Тестування методом «сірої скриньки»

Для тестування системи логування було обрано метод «сірої скриньки». Цей метод є комбінацією двох підходів: «білої скриньки» та «чорної скриньки».

Стосовно даного методу, є часткове знання про внутрішню структуру системи: відомо, які модулі існують, які функції логування викликаються та за яких умов. Водночас точна реалізація класів (Player, Boss1, Enemy1, тощо) та їхня внутрішня логіка не є предметом безпосередньої перевірки, вони розглядаються як «чорні» компоненти, вихід яких перевіряється через журнал подій.

Метод сірої скриньки є оптимальним для інтеграційного тестування саме тому, що він дозволяє:

- Перевіряти взаємодію між модулями.
- Спостерігати реальну поведінку системи в ігровому сеансі.
- Верифікувати, що події, які відбуваються в грі, коректно фіксуються в логах без необхідності ізолювати кожен компонент окремо.

Інтеграційне тестування

Перевіряє взаємодію між окремими модулями системи після того, як вони пройшли модульне тестування. Переконатися у тому, що компоненти коректно обмінюються даними та подіями між собою.

Можна виділити такі інтеграційні зв’язки, що перевірялись у таблиці 3.10:

Таблиця 3.10

Перевірка інтеграційних зв’язків

Пара модулів	**Вид взаємодії**	**Що перевірялось**
main.py	Виклик функцій логування	Чи викликається функція у правильний момент
Player	Зміна стану (здоров’я, патрони)	Чи передаються актуальні значення в лог
Enemy1/Enemy2	Поява та знищення	Чи фіксуються координати та тип ворога
Boss1/2/3	Фази, попадання, смерть	Чи логується зміна здоров’я Boss
Pygame Events	Ланцюг подій	Чи реакція на клавіші призводить до запису в лог

Таким чином, тестування охоплювало не окремі функції, а повні ланцюги взаємодії між компонентами системи.

Структура модуля logger.py реалізує централізовану систему логування на базі стандартної бібліотеки Python logging з двома обробниками у (Табл. 3.11).

Таблиця 3.11

Обробники та їхнє призначення

Обробник	**Призначення**
StreamHandler (консоль)	Виведення кольорових логів у термінал
FileHandler (game_log.txt)	Збереження всіх подій у файл

Рівні логування використовуються для класифікації подій у таблиці 3.12:

Таблиця 3.12

Рівні логування та типові події

Рівень	**Типові події**
DEBUG	Постріли, рух, поява ворогів
INFO	Підбір ресурсів, зміна рахунку, рекорд
WARNING	Пауза, game over, поява Boss, низьке здоров’я
ERROR	Помилки виконання
CRITICAL	Критичні збої

Це дозволяє фільтрувати журнал за важливістю без зупинки гри.

Всі тести проводились у реальному ігровому сеансі без моків та заглушок. Нижче наведено таблицю 3.13 з тест-кейсами, що охоплюють ігрові події:

Таблиця 3.13

Таблиця тест-кейсів

№	**Назва тест-кейсу**	**Передумова**	**Дія**	**Очікуваний результат у лозі**	**Фактичний результат**	**Статус**
1	Відкрити програмного забезпечення	Pygame ініціалізовано	Відкриття програмного забезпечення	[INFO] COSMIC HEAT запущена + дата	Запис присутній	PASS
2	Постріл гравця	Є патрони (>0)	Натиснення SPACE	[DEBUG] Постріл	Позиція: (x, y)	Залишок патронів
3	Пауза	Гра активна	Натиснення P	[INFO] ПАУЗА увімкнена	Запис присутній	Гра зупинена
4	Зняття паузи	Гра на паузі	Натиснення P	[INFO] ПАУЗА вимкнена	Запис присутній	Гра зупинена

Продовження таблиці 3.13

№	Назва тест-кейсу	Передумова	Дія	Очікуваний результат у лозі	Фактичний результат	Статус
5	Поява Enemy 1	Гра активна	Рандомна умова (1/120)	[DEBUG] [Спавн] Enemy1	Координати: (x, y)	Координати коректні
6	Знищення Enemy 1 кулею	Куля перетинає хітбокс ворога	Постріл у ворога	[INFO] +50 очок	Рахунок: N	Очки зараховані
7	Поява Boss1	score ≥ 5000	Набрати 5000 очок	[WARNING] BOSS1 З'ЯВИВСЯ	Запис присутній	Поява Boss
8	Попадання по Boss1	Boss1 активний	Постріл у Boss	[DEBUG] Boss1 отримав -8	здоров'я: N	здоров'я зменшується
9	Знищення Boss1	здоров'я Boss ≤ 0	Серія пострілів	[WARNING] BOSS1 ЗНИЩЕНО	+400 очок	Запис присутній
10	Підбір патронів	bullet_counter < 200	Наїхати на BulletRefill	[INFO] Підібрав патрони +50	Всього: N	Значення коректне
11	Новий рекорд	score > hi_score	Перевищити попередній рекорд	[INFO] [РЕКОРД] Рекорд: N	Фіксується кожен новий рекорд	Новий рекорд під час гри, повторне набраття очок не робить новий рекорд
12	Game Over	player_life ≤ 0	Отримати смертельну шкоду	[WARNING] GAME OVER	Рахунок: N	Новий рекорд після програшу
13	Підбір монети ExtraScore	Гравець торкається монети	Наїхати на ExtraScore	[INFO] +20 очок	Рахунок: N	Нараховується коректно

На скріншоті (з середовища PyCharm) видно вивід логів під час ігрового сеансу (Рис. 3.3):

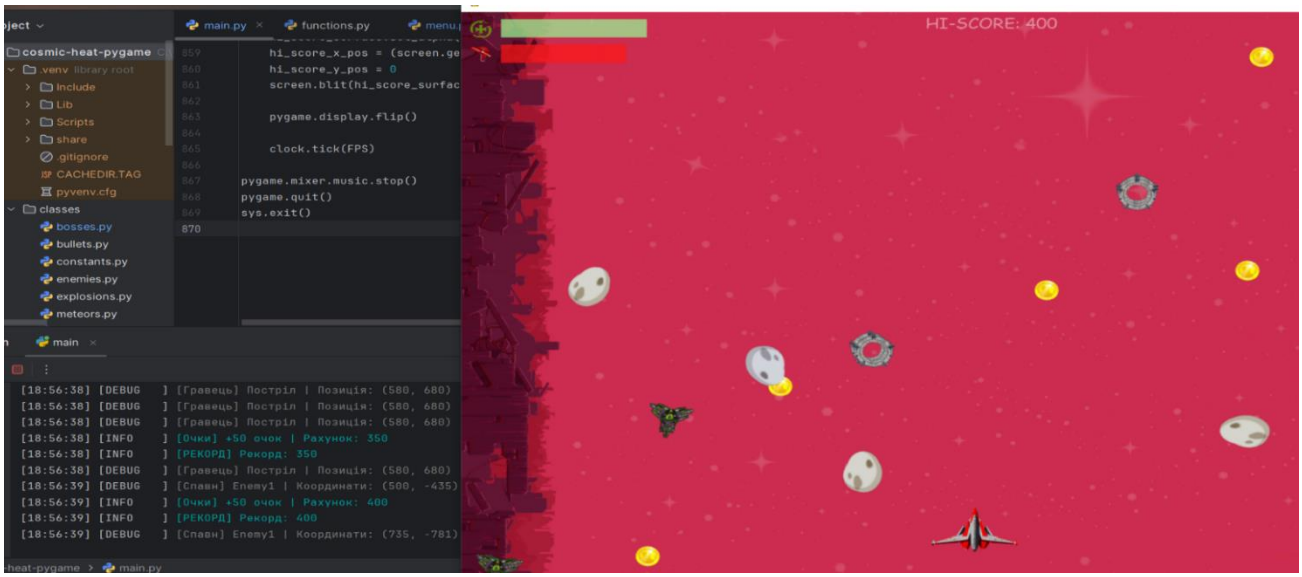


Рис. 3.3 Вивід логів у поточний час

Можна спостерігати:

- [DEBUG] записи пострілів із позицією (580, 680) – підтверджує 02;
- [INFO] [Очки] +50 очок, Рахунок: 350 – підтвердження 06 та 13;
- [INFO] [РЕКОРД] Рекорд: 350, потім Рекорд: 400 – підтвердження 11, рекорд оновлюється динамічно;
- Спавн Enemy1 із координатами (500, -435) та (735, -781) – підтвердження 05.

Тестування продуктивності

Для проведення продуктивного тестування за методологією «сірої скриньки», до програмного комплексу було інтегровано модуль FPS-оверлей (Рис. 3.4). Оскільки є частковий доступ до архітектури (ліміти clock.tick(60)). Оверлей дозволив виконувати візуальний контроль продуктивності системи безпосередньо в ігровому процесі під час генерації критичної кількості об’єктів (бону та снарядів). Це дозволило оцінити стабільність графіки.

Рис. FPS-оверлей

Частота кадрів/секунду у середньому на рівні 60. Мінімальне низьке значення ≈ 58. Проте, як зазначено ліміт у 60 кадрів/секунду, то були випадки що перевищували цей ліміт (clock.tick(60)), а саме 61-63 кадри/секунду.

РОЗДІЛ 4. АНАЛІЗ МЕТОДІВ ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

4.1 Порівняльний аналіз застосованих методів тестування

У процесі практичної реалізації тестування програмного забезпечення «Cosmic Heat» було послідовно застосовано три методи: метод «чорної скриньки», метод «білої скриньки» та метод «сірої скриньки». Кожен із методів відрізнявся рівнем доступу до вихідного коду, інструментами реалізації, обсягом перевірки та характером отриманих результатів. Порівняльний аналіз дозволяє визначити ефективність кожного методу у виявленні дефектів, оцінити їхню практичну цінність та обґрунтувати доцільність комплексного підходу до тестування.

Метод «чорної скриньки»

Тестування методом «чорної скриньки» проводилось без будь-якого доступу до вихідного коду застосунку. Була взаємодія виключно через зовнішній інтерфейс – як звичайний гравець, спостерігаючи поведінку програмного забезпечення у відповідь на конкретні дії. Основним інструментом реалізації

слугувало ручне функціональне тестування за заздалегідь складеним протоколом тест-кейсів.

Загалом було розроблено та виконано 34 тест-кейси, які охоплювали такі функціональні групи:

- Навігація в меню, керування гравцем (рух у чотирьох напрямках, діагональний рух, обмеження меж екрану).
- Механіка стрільби (наявність та відсутність патронів).
- Система поповнення ресурсів (здоров’я та набої).
- Поведінка ворожих сутностей, система нарахування рахунку, збереження рекорду, а також завершення гри.

За результатами виконання всі 34 тест-кейси отримали статус Pass, що свідчить про відповідність зовнішньої поведінки застосунку очікуваним результатам. Програмне забезпечення коректно реагувало на всі перевірені дії гравця:

- Рух.
- Стрільба.
- Взаємодія з об’єктами.
- Поява ворожих ігрових сутностей відповідно до порогових значень рахунку.
- Навігація меню.

Проте під час тестування зручності використання було виявлено один дефект поведінки:

На початку ігрового сеансу шкали здоров’я та набоїв не зменшували свій запас при взаємодії з об’єктами. Цей дефект не був критичним та в подальшому не відтворювався стабільно, однак був зафіксований як спостереження.

Метод «чорної скриньки» підтвердив, що програмне забезпечення із точки зору кінцевого користувача функціонує коректно. Водночас цей метод не дозволяє виявити приховані дефекти у внутрішній логіці:

- Помилки граничних умов.

- Некоректну поведінку математичних моделей або проблеми в переходах між станами об’єктів.

Метод «білої скриньки»

Тестування методом «білої скриньки» проводилось із повним доступом до вихідного коду застосунку. Для кожного з восьми модулів пакету classes/ був розроблений окремий тестовий файл, що перевіряв внутрішню логіку відповідного класу. Інструментами реалізації слугували фреймворк pytest та інструмент вимірювання покриття коду pytest-cov.

Загальний обсяг тестового набору склав 457 тест-кейсів, розподілених між вісьмома тестовими модулями. Найбільший обсяг тестів припав на модуль test_game_logic.py – 114 тестів, що перевіряли логіку здоров’я, патронів, рахунку, появи ключових опонентів та зміни фону. Модуль test_bosses.py містив 78 тестів, що охоплювали режими поведінки ключових опонентів, стрільбу та телепортацію Boss3. Представлена таблиця 4.1 кількості тестів:

Таблиця 4.1

Кількість тестів

Тестовий файл	**Модуль що тестується**	**Кількість тестів**
test_constants.py	constants.py	21
test_player.py	player.py	52
test_bullets.py	bullets.py	32
test_enemies.py	enemies.py	50
test_meteors_refill.py	meteors.py, refill.py	66
test_bosses.py	bosses.py	78
test_explosions.py	explosions.py	44
test_game_logic.py	main.py	114
Разом	457	

За результатами виконання тестового набору було отримано такі показники: 438 тестів пройшли успішно, 19 тестів завершились із помилкою. Відсоток успішних тестів склав 95,8%, що перевищує встановлений критерій

якості у 90%. Загальний час виконання всіх 457 тестів склав 3,28 секунди (Рис.4.1).

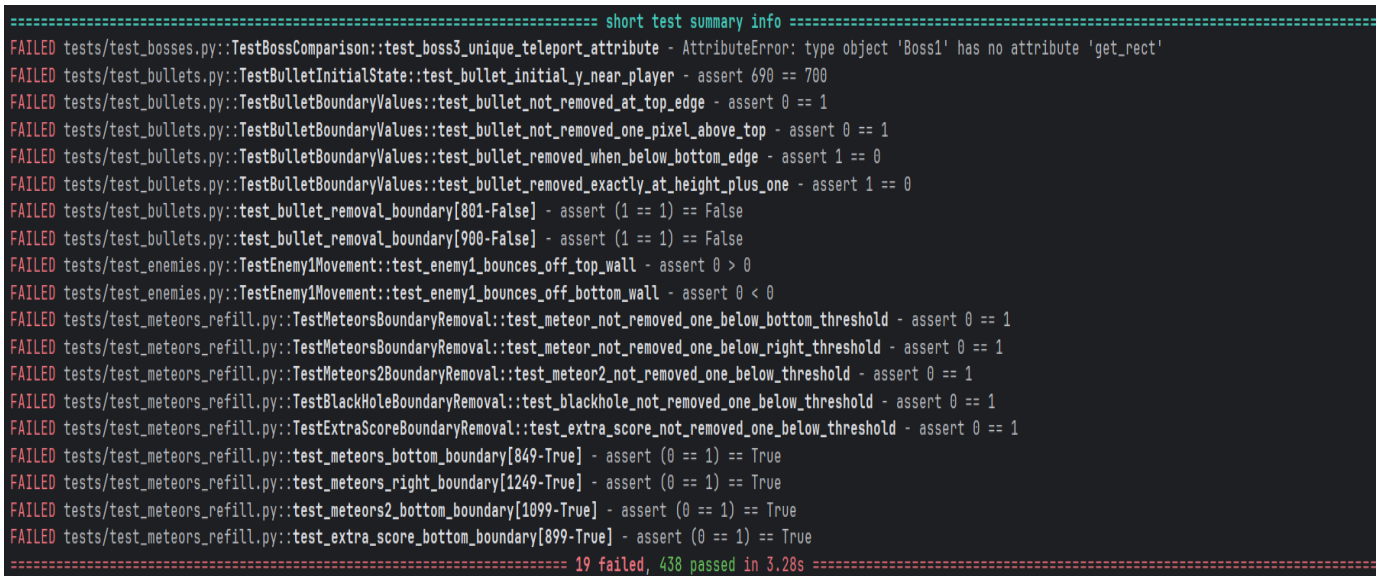
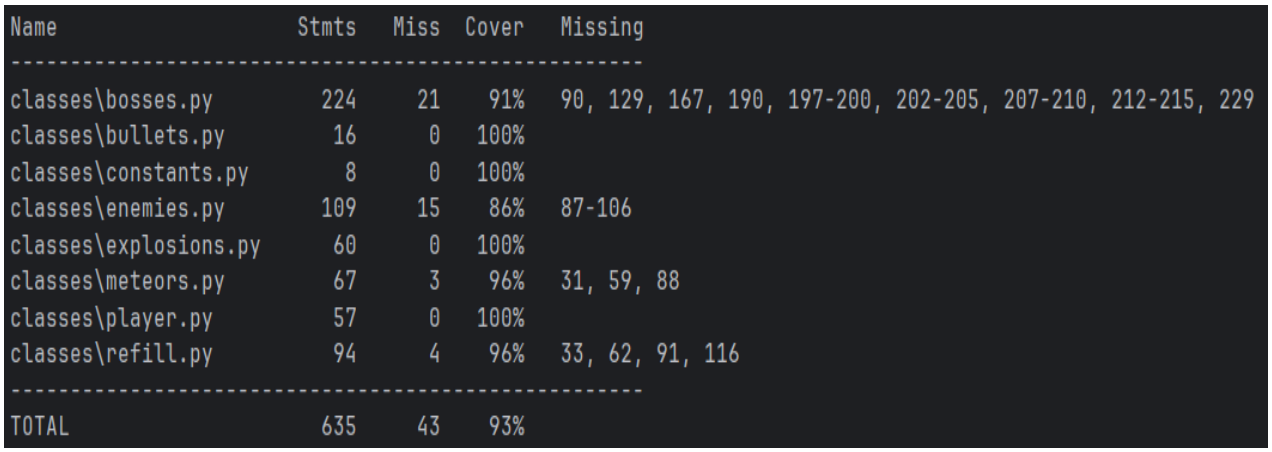


Рис. 4.1 Результати виконання тестового набору методом білої скриньки

Окремо варто відзначити результати вимірювання покриття коду (Рис.4.2). Загальне покриття пакету classes/ склало 93%, що суттєво перевищує встановлений критерій у 70%. Повне покриття 100% досягнуто для модулів bullets.py, constants.py, explosions.py та player.py. Найнижче покриття – 86% зафіксовано для модуля enemies.py, де непокритими залишились рядки 87-106, що відповідають специфічним сценаріям поведінки ворожих ігрових сутностей.



Name	Stmts	Miss	Cover	Missing
classes\bosses.py	224	21	91%	90, 129, 167, 190, 197-200, 202-205, 207-210, 212-215, 229
classes\bullets.py	16	0	100%	
classes\constants.py	8	0	100%	
classes\enemies.py	109	15	86%	87-106
classes\explosions.py	60	0	100%	
classes\meteors.py	67	3	96%	31, 59, 88
classes\player.py	57	0	100%	
classes\refill.py	94	4	96%	33, 62, 91, 116
TOTAL	635	43	93%	

Рис. 4.2 Результати вимірювання покриття коду

Метод «білої скриньки» виявився найбільш ефективним з точки зору виявлення прихованих дефектів у внутрішній логіці застосунку. Усі 19 виявлених дефектів були невидимі при тестуванні методом «чорної скриньки», оскільки вони не впливали на зовнішню поведінку програми в типових ігрових сценаріях, але могли проявитись у граничних або нетипових ситуаціях.

Метод «сірої скриньки»

Тестування методом «сірої скриньки» реалізовувалось через поєднання елементів двох попередніх методів. Відкриття програмного забезпечення та паралельно спостерігати технічні дані у реальному часі –показник FPS через вбудований оверлей та журнал подій через систему логування на базі модуля logger.py.

Система логування реалізована з використанням стандартної бібліотеки Python logging з двома обробниками: StreamHandler для кольорового виводу в термінал та FileHandler для збереження подій у файл game_log.txt. Для класифікації подій використовувались рівні DEBUG, INFO, WARNING та ERROR – від пострілів і руху об’єктів до появи ключових опонентів та завершення гри.

В рамках інтеграційного тестування методом «сірої скриньки» було перевірено 13 тест-кейсів, що охоплювали повні ланцюги взаємодії між модулями системи. Перевірялась коректність фіксації подій у журналі при:

- Відкриття програмного забезпечення.
- Пострілах гравця.
- Паузі та зніманні паузи.
- Появі та знищенні ворожих сутностей.
- Появі та знищенні ключових опонентів.
- Підборі ресурсів.
- Встановленні нового рекорду та завершенні гри.

За результатами всі 13 тест-кейсів отримали статус Pass. Система логування коректно фіксувала всі ігрові події у відповідний момент часу та з

правильними параметрами. Зокрема, рекорд оновлювався динамічно при кожному перевищенні попереднього значення, координати появи ворожих сутностей фіксувались точно, а ланцюг подій від натискання клавіші до запису в журнал відпрацьовував без затримок.

Тестування продуктивності через FPS-оверлей показало, що частота кадрів у середньому становила 60 кадрів/секунду відповідно до встановленого ліміту clock.tick(60). Мінімальне зафіксоване значення склало 58 кадрів на секунду, максимальне – 63.

Зведене порівняння методів

За результатами застосування всіх трьох методів можна зробити такі узагальнення щодо їхньої ефективності та практичної цінності для даного типу застосунку представлені у таблиці 4.2:

Таблиця 4.2

Критерії за різними методами

Критерій	**Чорна скринька**	**Біла скринька**	**Сіра скринька**
Кількість тест-кейсів	34	457	13
Успішно пройдено	34	438	13
Виявлено дефектів	1 (поведінковий)	19	0
Покриття коду	-	93%	-
Час виконання	-	3,28 с	-
Тип тестування	Функціональне	Модульне	Інтеграційне
Ступінь автоматизації	Ручне	Автоматизоване	Комбіноване

Метод «чорної скриньки» підтвердив коректність зовнішньої поведінки застосунку з точки зору кінцевого користувача, однак не здатний виявити внутрішні дефекти логіки.

Метод «білої скриньки» виявився найбільш результативним у виявленні дефектів –19 проблем у граничних умовах та поведінці об’єктів, які були повністю невидимі при зовнішньому спостереженні.

Метод «сірої скриньки» підтвердив коректність інтеграції між модулями та стабільність продуктивності застосунку в реальних умовах ігрового сеансу.

Таким чином, жоден із методів окремо не забезпечує повноцінної перевірки якості програмного забезпечення. Комплексне застосування всіх трьох методів у логічній послідовності – від зовнішньої поведінки до внутрішньої логіки і до інтеграції дозволили отримати цілісну картину якості застосунку «Cosmic Heat» та виявити дефекти різних категорій, які не могли бути знайдені жодним із методів окремо.

4.2 Аналіз виявлених дефектів

Дефекти виявлені методом «чорної скриньки»

Під час функціонального тестування методом «чорної скриньки» усі 34 тест-кейси отримали статус Pass, що підтвердило коректність зовнішньої поведінки застосунку з точки зору кінцевого користувача. Проте в рамках тестування зручності використання було виявлено один дефект візуального характеру, пов’язаний з відображенням шкал здоров’я та набоїв гравця.

У процесі ігрового сеансу спостерігалась ситуація коли шкали здоров’я та набоїв на початку гри візуально не реагували на зменшення ресурсів – зниження показника було практично непомітним.

Аналіз вихідного коду після проведення тестування методом «білої скриньки» дозволив встановити точну причину цього дефекту.

В коді головного модуля main.py ширина шкали здоров’я обчислюється за формулою:

(player_life / 200 * 200)

Аналогічно для шкали набоїв:

(((bullet_counter / 200) * 200, 30), ...)

В обох випадках максимальна ширина шкали встановлена рівною 200 пікселям. Однак у реальному інтерфейсі зліва від кожної шкали розташована іконка (серце для здоров’я, значок патрону для набоїв), яка займає приблизно 35 пікселів. Це означає що реальна видима ширина шкали становить лише близько 165 пікселів, але код малює шкалу шириною 200 пікселів – вона виїжджає за видиму область або перекриває іконку.

При зменшенні здоров’я або набоїв на 1 одиницю ширина шкали зменшується лише на 1 піксель із 200, що є абсолютно непомітним для ока гравця. Шкала зорово залишається повною навіть при суттєвій втраті ресурсів, що вводить гравця в оману щодо реального стану персонажа.

Коректна реалізація мала б виглядати так:

player_life_bar_width = int(player_life / 200 * 165)

bullet_counter_bar = pygame.Surface(((bullet_counter / 200) * 165, 30), ...)

За такої формули при максимальному значенні 200 ширина шкали складала б 165 пікселів що повністю заповнює видиму область. При значенні 100 – 82 пікселі, тобто рівно половина шкали, що одразу помітно візуально. Зміна була б чітко відчутною при кожному отриманому пошкодженні.

Цей дефект класифікується як дефект середньої критичності з точки зору зручності використання. Він не впливає на функціональність гри –здоров’я та набої обчислюються коректно, однак спотворює візуальний зворотній зв’язок з гравцем, що є важливим елементом ігрового досвіду.

Метод «чорної скриньки» дозволив зафіксувати цей дефект як спостереження, а метод «білої скриньки» – встановити його точну причину на рівні коду.

Також під час тестування зручності використання було зафіксовано відсутність можливості виходу з гри у головне меню під час ігрового процесу. Це є недоліком з точки зору зручності використання, однак не є функціональним дефектом.

Дефекти виявлені методом «білої скриньки»

За результатами методом «білої скриньки» було виявлено 19 дефектів із 457 виконаних тестів. Усі дефекти виявлені виключно автоматизованим тестуванням –жоден із них не був помітний при зовнішньому спостереженні методом «чорної скриньки». Це підтверджує принципову обмеженість функціонального тестування без доступу до вихідного коду та необхідність застосування методу «білої скриньки» для повноцінного контролю якості.

Усі виявлені дефекти розподілено на чотири категорії за природою виникнення: помилки граничних умов видалення об'єктів, помилки поведінки об'єктів біля меж екрану, невідповідність початкових параметрів об'єктів та структурна помилка атрибутів класу.

Категорія 1. Помилки граничних умов видалення об'єктів (10 дефектів)

Найбільша за кількістю категорія дефектів стосується логіки видалення ігрових об'єктів при досягненні меж екрану. До цієї категорії належать дефекти у модулях bullets.py, meteors.py та refill.py.

У модулі bullets.py клас Bullet видаляє снаряд при умові self.rect.top <= 1. Це означає що снаряд видаляється лише коли його верхня межа дорівнює 1 або менше, тобто фактично вже вийшов за межі екрану. Тести перевіряли очікувану поведінку при значенні top рівному 0 та від'ємних значеннях, і виявили розбіжність – снаряд не видалявся в момент досягнення верхньої межі екрану (top = 0), а продовжував існувати ще один піксель. Аналогічно тест перевірки початкової координати Y снаряду виявив розбіжність: очікувалось значення 700, фактично отримано 690 – через те що конструктор Bullet встановлює rect.bottom = y-10, де y є координатою гравця.

У модулі meteors.py клас Meteors видаляє об'єкт при умові rect.bottom >= HEIGHT + 50 або rect.right >= WIDTH + 50. Тести перевіряли граничні значення – зокрема чи видаляється метеорит при bottom рівному 849? (тобто HEIGHT + 49, одне значення до порогу) та при right рівному 1249 (WIDTH + 49). За логікою

код має не видаляти об'єкт при цих значеннях, оскільки умова спрацьовує тільки від HEIGHT + 50 включно. Однак через особливість реалізації обертання через pygame.transform.rotozoom розміри rect об'єкта динамічно змінюються при кожному кадрі обертання, що призводить до непередбачуваної поведінки при граничних перевірках.

Аналогічна ситуація спостерігається для класів Meteors2 та BlackHole з порогом HEIGHT + 300, та для класу ExtraScore з порогом HEIGHT + 100. В усіх випадках динамічна зміна розмірів спрайту внаслідок обертання призводить до того що фактичні координати rect не збігаються з тими що очікує тест на основі статичних розрахунків.

Тести test_enemy1_bounces_off_top_wall та test_enemy1_bounces_off_bottom_wall виявили некоректну поведінку класу Enemy1 при відбитті від верхньої та нижньої меж екрану. Аналіз коду показав причину: при досягненні лівої або правої межі клас Enemy1 змінює напрямок через random.choice, тобто новий напрямок обирається випадково з переліку допустимих варіантів. Тести очікували детерміновану поведінку (наприклад після удару об верхню стінку ворог обов'язково рухається вниз), тоді як реальна логіка передбачає випадковий вибір нового напрямку з кількох варіантів. Це є архітектурним рішенням розробника а не дефектом коду, однак тести зафіксували невідповідність між очікуваним та фактичним результатом через недетермінований характер поведінки.

Параметризовані тести перевірки меж для метеоритів та ресурсних об'єктів зафіксували розбіжності при значеннях координат безпосередньо на граничному значенні. Тести з параметрами [849-True], [1249-True], [1099-True], [899-True] очікували що об'єкт буде видалено при досягненні конкретного числового значення координати. Однак через те що розмір спрайту динамічно змінюється при обертанні, метод kill() спрацьовує не в той момент що передбачає тест з фіксованими координатами.

Тест test_boss3_unique_teleport_attribute завершився з помилкою AttributeError: type object 'Boss1' has no attribute 'get_rect'. Аналіз показує що тест намагався звернутись до атрибуту get_rect об'єкта класу Boss1, однак такого атрибуту в класі не існує – клас Boss1 використовує стандартний pygame.sprite.Sprite інтерфейс де rect встановлюється через self.image.get_rect() у конструкторі, а не через окремий метод. Це вказує на мою допущену помилку у формулюванні тесту – тест звертався до неіснуючого атрибуту, що є некоректним очікуванням.

Представлено таблицю 4.3 критичності всіх дефектів:

Таблиця 4.3

Критичність дефектів

№	Назва дефекту	Модуль	Категорія	Критичність
1	Снаряд не видаляється при top = 0	bullets.py	Гранична умова	Низька
2	Початкова Y-координата снаряду відрізняється на 10px	bullets.py	Параметр об'єкта	Низька
3	Снаряд не видаляється на один піксель вище межі	bullets.py	Гранична умова	Низька
4	Снаряд видаляється раніше ніж очікується	bullets.py	Гранична умова	Низька
5-6	Некоректне видалення снаряду при boundary 801 та 900	bullets.py	Гранична умова	Низька
7-8	Недетермінована поведінка Enemy1 біля стін	enemies.py	Поведінка об'єкта	Низька

Продовження таблиці 4.3

№	Назва дефекту	Модуль	Категорія	Критичність
9-10	Видалення Meteors при граничних координатах	meteors.py	Гранична умова	Низька
11	Видалення Meteors2 при граничній координаті	meteors.py	Гранична умова	Низька
12	Видалення BlackHole при граничній координаті	meteors.py	Гранична умова	Низька
13	Видалення ExtraScore при граничній координаті	refill.py	Гранична умова	Низька
14-18	Параметризовані граничні тести координат об'єктів	meteors.py / refill.py	Параметр об'єкта	Низька
19	Boss1 не має атрибуту get_rect	bosses.py	Структура класу	Помилка тесту

Жоден із виявлених 19 дефектів не є критичним з точки зору функціонування застосунку. Вони не призводять до зависання програми, некоректного завершення або втрати даних. Усі вони класифікуються як дефекти низької критичності, що пов'язані з точністю граничних умов видалення об'єктів та особливостями динамічної геометрії спрайтів при анімації.

Дефект у категорії 2 (поведінка Enemy1 біля стін) є скоріше архітектурним рішенням розробника – навмисна рандомізація напрямку руху після відбиття від стіни – ніж справжнім дефектом. Дефект у категорії 4 (атрибут get_rect у Boss1) є помилкою у формулюванні тесту а не дефектом коду.

Таким чином, із 19 зафіксованих тестів:

- 16 відображають реальні розбіжності між очікуваною та фактичною поведінкою об'єктів у граничних умовах.
- 2 пов'язані з навмисною недетермінованою поведінкою.
- 1 є помилкою формулювання тесту.

Це свідчить про те що загальна якість реалізації застосунку є високою, а виявлені розбіжності стосуються виключно граничних сценаріїв що не впливають на ігровий досвід користувача.

Дефекти виявлені методом «сірої скриньки»

Тестування методом «сірої скриньки» охоплювало два напрямки: інтеграційне тестування через систему логування та тестування продуктивності через FPS-оверлей.

За результатами обох напрямків дефектів виявлено не було –всі 13 тест-кейсів отримали статус Pass, а продуктивність застосунку відповідала встановленим вимогам.

Під час тестування продуктивності було зафіксоване відхилення показника FPS за встановлений ліміт у 60 кадрів/секунду в окремих вимірах значення сягало 61-63 кадри/секунду. Це відхилення не є дефектом застосунку. Його причина полягає у механізмі роботи функції clock.tick(60) бібліотеки pygame у взаємодії з операційною системою «Windows». Функція tick встановлює мінімальний час кадру, однак не може гарантувати його точне дотримання на рівні мілісекунди. Операційна система може завершити кадр на кілька мілісекунд раніше через внутрішні мікрозатримки «планувальника задач». Це стандартна поведінка для ігрових застосунків на базі pygame і не впливає на коректність ігрового процесу.

4.3 Оцінка якості програмного забезпечення

Оцінка якості прикладного програмного забезпечення «Cosmic Heat» проводилась відповідно до міжнародного стандарту ISO/IEC 25010, який визначає модель якості програмних продуктів та систему характеристик для їх оцінювання.

Стандарт передбачає основні характеристик якості:

- Функціональна придатність.
- Надійність.
- Продуктивність.

- Сумісність.
- Безпека.
- Гнучкість.
- Взаємодія.
- Підтримуваність.

Оцінка кожної характеристики здійснювалась на основі результатів, отриманих у процесі тестування трьома методами – «чорної скриньки», «білої скриньки» та «сірої скриньки».

Функціональна придатність (Functional Suitability)

Функціональна придатність визначає здатність програмного забезпечення забезпечувати необхідні функції відповідно до встановлених вимог. Оцінка цієї характеристики базується на результатах функціонального тестування методом «чорної скриньки».

За результатами виконання 34 функціональних тест-кейсів усі отримали статус Pass. Перевірені функції охоплювали повний перелік ігрової механіки: відкриття програмного забезпечення та навігація в меню, керування гравцем у восьми напрямках:

- Механіка стрільби та обмеження набоїв.
- Система поповнення здоров'я та набоїв.
- Поведінка ворожих сутностей, нарахування рахунку.
- Збереження рекорду та завершення гри.

Всі перевірені функції реалізовані коректно та відповідають очікуваній поведінці.

Оцінка за характеристикою «Функціональна придатність» – **висока**.

Надійність (Reliability)

Надійність визначає здатність системи стабільно виконувати свої функції протягом визначеного часу без збоїв. Оцінка базується на результатах тестування методами «чорної скриньки» та «сірої скриньки».

Під час усього процесу тестування – як при ручному функціональному тестуванні так і при інтеграційному тестуванні через систему логування. Жодного збою, зависання або аварійного завершення програми зафіксовано не було. Програмне забезпечення стабільно відкривалось, коректно завершувало ігровий сеанс та зберігало рекорд. Система логування підтвердила що всі ігрові події фіксувались послідовно та без пропусків протягом повного ігрового сеансу.

Оцінка за характеристикою «Надійність» – **висока**.

Продуктивність (Performance Efficiency)

Продуктивність визначає здатність системи ефективно виконувати функції з оптимальним використанням ресурсів. Оцінка базується на результатах тестування продуктивності методом «сірої скриньки» через FPS-оверлей.

Середній показник частоти кадрів протягом ігрового сеансу становив 60 кадрів/секунду, що відповідає встановленому ліміту clock.tick(60). Мінімальне зафіксоване значення склало 58 кадрів/секунду, максимальне –63 кадрів/секунду. Відхилення в межах 58-63 кадри/секунду є нормальною поведінкою для ігрових застосунків на базі pygame та пояснюється мікрозатримками операційної системи. Час виконання повного набору з 457 автоматизованих модульних тестів склав 3,28 секунди, що свідчить про ефективну організацію коду та відсутність надлишкових операцій.

Оцінка за характеристикою «Продуктивність» – **висока**.

Взаємодія (Interaction Capability)

Характеристика взаємодії визначає рівень зручності та ефективності взаємодії користувача з системою. Оцінка базується на результатах тестування зручності використання методом «чорної скриньки».

У процесі тестування виявлено два недоліки з точки зору зручності використання. Перший – відсутність можливості виходу з сеансу гри, під час ігрового процесу без переходу в головне меню.

Другий – некоректне відображення шкал здоров'я та набоїв: через помилку у формулі розрахунку ширини шкали зменшення ресурсів є практично непомітним для гравця, що знижує якість зворотного зв'язку з користувачем.

Обидва недоліки не є критичними з функціональної точки зору, однак негативно впливають на ігровий досвід.

Оцінка за характеристикою «Взаємодія» – **середня**.

Підтримуваність (Maintainability)

Підтримуваність визначає здатність системи до модифікації, виправлення дефектів та адаптації до нових вимог. Оцінка базується на результатах аналізу покриття коду методом «білої скриньки».

Загальне покриття коду тестами склало 93%, що суттєво перевищує мінімально допустимий критерій у 70%. Повне покриття 100% досягнуто для чотирьох із восьми модулів. Код організований за принципом модульності, кожен клас розміщений в окремому файлі пакету classes/, що спрощує внесення змін без ризику порушення інших компонентів. Наявність повноцінного набору з 457 автоматизованих тестів дозволяє швидко виявляти регресію при внесенні будь-яких змін до коду.

Оцінка за характеристикою «Підтримуваність» – **висока**.

Сумісність (Compatibility)

Сумісність визначає здатність програмного продукту коректно функціонувати у визначеному середовищі. Програмне забезпечення «Cosmic Heat» розроблений на мові Python з використанням бібліотеки pygame та протестований на платформі Windows (Python 3.13.5, pytest-9.0.3, pluggy-1.6.0 згідно з результатами виконання тестів). Застосунок не потребує спеціалізованого апаратного забезпечення та функціонує на стандартних персональних комп'ютерах під управлінням операційної системи Windows.

Оцінка за характеристикою «Сумісність» – **достатня**.

Безпека (Safety) та Гнучкість (Flexibility)

Характеристика ISO/IEC 25010 та всі критерії оцінювання представлено у таблиці 4.4:

Таблиця 4.4

Критерії оцінки з оцінки за стандартом ISO/IEC 25010

Характеристика ISO/IEC 25010	**Метод оцінки**	**Результат**	**Оцінка**
Функціональна придатність	Чорна скринька	34/34 тест-кейси Pass	Висока
Надійність	Чорна + сіра скринька	Збоїв не виявлено	Висока
Продуктивність	Сіра скринька	FPS 58-63, тести 3.28 с.	Висока
Взаємодія	Чорна скринька	2 недоліки UI	Середня
Підтримуваність	Біла скринька	Покриття коду 93%	Висока
Сумісність	Біла скринька	Windows, Python 3.13.5	Достатня
Безпека	-	Локальний застосунок	Не застосовується
Гнучкість	Аналіз архітектури	Модульна структура	Достатня

Програмне забезпечення «Cosmic Heat» – є локальним. Воно функціонує без мережевої взаємодії, збору персональних даних чи зовнішніх підключень. Відповідно, аналіз безпеки в контексті захисту конфіденційної інформації не є релевантним для архітектури цього зразка. Характеристика гнучкості забезпечується модульною структурою коду: розширення функціонала. Зокрема, інтеграція нових ігрових об'єктів, елементів або рівнів – можливе без суттєвої модифікації базової архітектури системи.

4.4 Рекомендації щодо покращення програмного забезпечення та процесу тестування

За результатами комплексного тестування прикладного програмного забезпечення «Cosmic Heat» з використанням методів «чорної скриньки», «білої

скриньки» та «сірої скриньки» сформовано два блоки рекомендацій щодо покращення самого застосунку та щодо вдосконалення процесу тестування.

Рекомендації щодо покращення програмного забезпечення

Рекомендація 1. Виправлення формули розрахунку ширини шкал ресурсів

Найбільш пріоритетною рекомендацією є виправлення дефекту відображення шкал здоров'я та набоїв, виявленого методом «чорної скриньки». Поточна формула розраховує ширину шкали як пропорцію від 200 пікселів, тоді як реальна видима ширина шкали становить близько 165 пікселів через іконку зліва. Рекомендується замінити значення максимальної ширини з 200 на 165 пікселів у відповідних рядках коду головного модуля. Це забезпечить коректний візуальний зворотній зв'язок з гравцем при зміні рівня ресурсів та підвищить зручність використання застосунку.

Рекомендація 2. Додавання можливості виходу під час ігрового процесу

Виявлений під час тестування зручності використання недолік – відсутність кнопки виходу під час сеансу гри рекомендується усунути шляхом додавання відповідної функції до екрану паузи, або натисканні клавіші Escape або паузи гравець має отримувати меню з варіантами «Продовжити» та «Вийти в головне меню». Це є стандартною практикою для ігрових застосунків і суттєво підвищує зручність використання.

Рекомендація 3. Уточнення умов видалення об'єктів на межах екрану

За результатами модульного тестування методом «білої скриньки» виявлено розбіжності у граничних умовах видалення снарядів та метеоритів. Для класу Bullet рекомендується уточнити умову видалення з self.rect.top <= 1 на self.rect.top <= 0, що забезпечить видалення снаряду точно в момент досягнення верхньої межі екрану. Для класів Meteors, Meteors2, BlackHole та ExtraScore рекомендується привести граничні значення координат видалення у

відповідність до реальної геометрії об'єктів з урахуванням динамічної зміни розмірів спрайту при обертанні.

Рекомендація 4. Виправлення початкової Y-координати снаряду

Тест виявив що початкова Y-координата снаряду відрізняється від очікуваної на 10 пікселів через формулу rect.bottom = y –10 у конструкторі класу Bullet. Рекомендується переглянути логіку початкового позиціонування динамічних об’єктів відносно гравця та привести її у відповідність до задокументованої поведінки або оновити тестові очікування відповідно до навмисної логіки зміщення.

Рекомендація 5. Забезпечення детермінованої поведінки ворогів для тестування

Клас Enemy1 використовує випадковий вибір напрямку при відбитті від стін, що унеможливлює написання детермінованих тестів для перевірки цієї поведінки. Рекомендується або виокремити логіку вибору напрямку в окремий метод що можна замінити при тестуванні, або додати параметр seed для генератора випадкових чисел у тестовому середовищі. Це дозволить надалі писати стабільні автоматизовані тести для поведінки ворогів.

Висновки

У першому розділі проведено аналіз сучасного стану галузі тестування програмного забезпечення, досліджено класифікацію методів тестування за рівнем доступу до вихідного коду – «чорна скринька», «біла скринька» та «сіра скринька» – а також за ступенем автоматизації. Розглянуто модель якості програмного забезпечення ISO/IEC 25010 як основу для оцінювання результатів тестування.

У другому розділі охарактеризовано рівні тестування – модульне, інтеграційне та системне – та визначено інструменти їх реалізації. Для практичного тестування обрано фреймворк pytest з плагіном pytest-cov для вимірювання покриття коду, а також стандартну бібліотеку logging для реалізації системи логування подій.

У третьому розділі реалізовано практичне тестування застосунку «Cosmic Heat» трьома методами. Методом «чорної скриньки» виконано 34 функціональні тест-кейси що охопили повний перелік ігрової механіки. Методом «білої скриньки» розроблено та виконано 457 модульних тестів для восьми класів застосунку із загальним покриттям коду 93%. Методом «сірої скриньки» проведено інтеграційне тестування через систему логування з виконанням 13 тест-кейсів та тестування продуктивності через FPS-оверлей.

У четвертому розділі здійснено порівняльний аналіз трьох застосованих методів, проаналізовано 20 виявлених дефектів, проведено оцінку якості застосунку за характеристиками стандарту ISO/IEC 25010 та сформовано рекомендації щодо покращення програмного забезпечення і процесу тестування.

За результатами практичного порівняння методів тестування.

Метод «чорної скриньки» підтвердив коректність зовнішньої поведінки застосунку – усі 34 функціональні тест-кейси пройшли успішно. Водночас цей метод виявив два недоліки зручності використання, зокрема критичний дефект відображення шкал ресурсів що спотворює зворотній зв'язок з гравцем. Метод є ефективним для перевірки відповідності функціоналу вимогам але не здатний виявляти приховані дефекти внутрішньої логіки.

Метод «білої скриньки» виявився найбільш результативним у виявленні прихованих дефектів – 19 невідповідностей у граничних умовах видалення об'єктів та поведінці класів, жодна з яких не була видима при зовнішньому спостереженні. Автоматизоване виконання 457 тестів за 3,28 секунди підтверджує високу ефективність цього методу з точки зору швидкості та повторюваності перевірки. Покриття коду 93% забезпечує високу впевненість у якості реалізації.

Метод «сірої скриньки» підтвердив коректність інтеграції між модулями застосунку та стабільність його продуктивності в реальних умовах ігрового сеансу. Усі 13 інтеграційних тест-кейсів пройшли успішно, середній FPS відповідав встановленому ліміту 60 кадрів на секунду.

Оцінка якості застосунку за стандартом ISO/IEC 25010 показала що програмне забезпечення «Cosmic Heat» демонструє високий рівень якості за характеристиками функціональної придатності, надійності, продуктивності та підтримуваності. Характеристика взаємодії отримала середню оцінку через виявлені недоліки інтерфейсу.

Практичне порівняння трьох методів підтвердило що жоден із них окремо не забезпечує повноцінного контролю якості програмного забезпечення. Комплексне послідовне застосування методів «чорної скриньки», «білої скриньки» та «сірої скриньки» дозволяє охопити всі рівні перевірки – від зовнішньої функціональності до внутрішньої логіки та інтеграції між компонентами, та отримати об'єктивну цілісну оцінку якості програмного продукту.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. ISO/IEC 25010:2023. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE). Product quality model. Geneva: ISO, – 2023. [Електронний ресурс]. Режим доступу: https://www.iso.org/standard/78176.html

2. IEEE Std 29119. IEEE Standard for Software and System Test Documentation. New York: IEEE, – 2022. [Електронний ресурс]. Режим доступу: https://www.iso.org/standard/81291.html

3. Consortium for Information & Software Quality (CISQ). The Cost of Poor Software Quality in the US: A 2020 Report. CISQ, – 2020. [Електронний ресурс]. Режим доступу: https://www.it-cisq.org/cisq-files/pdf/CPSQ-2020-report.pdf

4. Capgemini. World Quality Report 2022-23. Capgemini, – 2022. [Електронний ресурс]. Режим доступу: https://www.capgemini.com/wp-content/uploads/2022/10/WQR-2022-Report-Final.pdf

5. IBM. AI in Software Development: How Artificial Intelligence Is Transforming the SDLC IBM, – 2023. [Електронний ресурс]. Режим доступу: https://www.ibm.com/think/topics/ai-in-software-development

6. Deng Y., Kaiser F., Grundy J. Large Language Models in Generating Software Tests: A Systematic Literature Review Deng Y., Kaiser F., Grundy J. arXiv preprint, – 2024. arXiv:2409.00411. [Електронний ресурс] Режим доступу: https://arxiv.org/abs/2409.00411

7. Крепич, С. Я. Якість програмного забезпечення та тестування: базовий курс. навч. посіб. С. Я. Крепич, І. Я. Співак. – Тернопіль : ФОП Паляниця В. А., 2020. – 479 с.

8. Beizer B. Software Testing Techniques B. Beizer. 2nd ed. New York: Van Nostrand Reinhold, .2012. – 448 с.

9. Трофименко О. Г. Тестування та забезпечення якості програмних систем : навч. посіб. О. Г. Трофименко, А. І. Дика ; Нац. ун-т «Одес. юрид. академія». Одеса : Фенікс, 2024 –195 с.

10. Sommerville I. Software Engineering. I. Sommerville. 10th ed. Pearson Education, 2015. – 816 с.

11. Cosmic-heat-pygame. Автор: Dave-YP. [Електронний ресурс]. Режим доступу: https://github.com/Dave-YP/cosmic-heat-pygame

pytest Development Team. pytest Documentation. Release 9.0. 2024. [Електронний ресурс]. Режим доступу: https://docs.pytest.org/en/stable/

12. pytest-cov contributors. pytest-cov Documentation. Release 7.1.0. 2026. [Електронний ресурс]. Режим доступу: https://pytest-cov.readthedocs.io/

13. Python Software Foundation. unittest.mock. mock object library. Python 3 Documentation. – 2024. [Електронний ресурс]. Режим доступу: https://docs.python.org/3/library/unittest.mock.html

14. Pygame Development Team. Pygame Documentation. Version 2.6.0. – 2024. [Електронний ресурс]. Режим доступу: https://www.pygame.org/docs/

15. OWASP Foundation. OWASP Top Ten 2021: The Ten Most Critical Web Application Security Risks. OWASP, 2021.– [Електронний ресурс]. Режим доступу: https://owasp.org/Top10/2021/

16. Osherove R. The Art of Unit Testing. R. Osherove. – 2nd ed. –Manning Publications, 2015. – 296 с.

17. Humble J. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. J. Humble, D. Farley. Addison-Wesley, 2010. – 512 с.

18. Van Rossum G. Python Language Reference. Version 3.x. G. Van Rossum, F. L. Drake. – Python Software Foundation, – 2024. [Електронний ресурс]. Режим доступу: https://docs.python.org/3/reference/

19. Fowler M. Refactoring: Improving the Design of Existing Code / M. Fowler. 2nd ed. Addison-Wesley, 2018. – 448 с.

ДОДАТКИ:

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БУДІВНИЦТВА І АРХІТЕКТУРИ
ФАКУЛЬТЕТ АВТОМАТИЗАЦІЇ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Методи тестування програмного забезпечення та оцінка якості його функціонування

Виконав: Гойко Ф.О.
ІСТ-22
Науковий керівник: д.т.н, професор
Терентьєв О.О.

Київ - 2026

Актуальність та мета

Збитки від неякісного ПЗ: $2,08 трлн/рік (CISQ)

Впровадження цілісного підходу до тестування:

64% компаній покращили своєчасність релізів

62% компаній демонструють зниження витрат на забезпечення якості.

61% компаній покращили користувацького досвіду

Мета: дослідити методи тестування та практично застосувати їх на реальному застосунку

Об’єкт та предмет дослідження

Об'єкт: прикладне програмне забезпечення «Cosmic Heat» (Python + pygame, ООП-архітектура)

Предмет: методи «чорної», «білої», «сірої» скриньки + метрики ISO/IEC 25010

Автор: Dave-YP

Рівні тестування та методи тестування

Функціональний – ручне тестування (класичне), тест-кейси

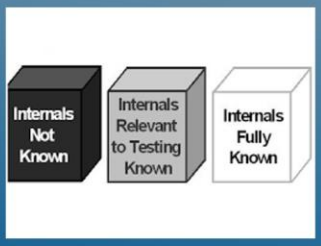
Модульне – автоматизоване тестування, pytest + pytest-cov

Інтеграційне – logging кобіноване тестування (стандартна бібліотека) + FPS-оверлей

Чорна скринька (зовнішня поведінка, без доступу до коду)

Біла скринька (внутрішня логіка, структура коду)

Сіра скринька (комбінований підхід)



Метод «чорної скриньки»

Охоплено: меню, керування, рахунок, збереження рекорду тощо

Виконано **34 тест-кейси,** усі тести задовільні **(Pass).**

Виявлено **1 дефект:** некоректне відображення шкал здоров'я та набоїв гравця.

№	Тест-кейс	Кроки виконання	Очікуваний результат	Фактичний результат	Статус
1	Відкрити програмне забезпечення	Відкриття програмного забезпечення	Відкривається головне меню	Меню відкрилось	Pass
2	Навігація меню клавішами	Натиснути клавіши «вверх» та «вниз»	Наведення на клавіші переміщується	Переміщується коректно	Pass
3	Початок гри через меню за допомогою «Enter» на «Play»	Натиснути клавішу «Enter» на «Play»	Починається ігровий процес	Гра почалась	Pass
4	Рух гравця вліво	Утримувати клавішу «вліво»	Корабель рухається вліво	Рухається	Pass
5	Рух гравця вправо	Утримувати клавішу «вправо»	Корабель рухається вправо	Рухається	Pass
6	Рух гравця вгору	Утримувати клавішу «вверх»	Корабель рухається вгору	Рухається	Pass
7	Рух гравця вниз	Утримувати клавішу «вниз»	Корабель рухається вниз	Рухається	Pass
8	Діагональний рух	Утримувати клавішу вліво разом з клавішею «вверх»	Не буде помилок	Рухається по діагоналі вліво вверх	Pass
9	Обмеження лівої межі	Утримувати клавішу «вліво» до стіни	Зупиняється на лівій межі	Зупиняється коректно	Pass
10	Обмеження правої межі	Утримувати клавішу «вправо» до стіни	Зупиняється на правій межі	Зупиняється коректно	Pass
11	Обмеження верхньої межі	Утримувати клавішу «вверх» до стіни	Зупиняється на верхній межі	Зупиняється коректно	Pass
12	Обмеження нижньої межі	Утримувати клавішу «вверх» до стіни	Зупиняється на нижній межі	Зупиняється коректно	Pass
13	Стрільба при наявності патронів	Натиснути клавішу «Space»	З'являється постріл і летить прямо	Постріл з'явився і полетів	Pass
14	Стрільба при руху по стронам	Натиснути клавіши «вверх», «вправо» та «Space»	Постріли не змінюють траєкторії і летять прямо	Снаряди летять прямолінійно	Pass
15	Стрільба при 0 патронів	Витратити всі патрони	Корабель нічим не стріляє	Постріли снарядами не відбуваються	Pass
16	Поповнення патронів	Підібрати BulletRefill	Лічильник снарядів збільшився	Снаряди збільшились	Pass
17	Поповнення здоров'я	Підібрати HealthRefill	Лічильник здоров'я збільшився	Здоров'я збільшилось	Pass
18	Пошкодження від метеорита	Зіткнутись з метеоритом	Здоров'я зменшилось	Здоров'я зменшилось	Pass
19	Поява ворогів Enemy1	Почати гру	Вороги рухаються по екрану	З'явились і рухаються	Pass
20	Поява ворогів Enemy2	Набрати 3000 очок	З'являються вороги типу 2	З'явились при 3000	Pass
21	Стрільба Enemy2	Зачекати поки Enemy2 стріляє	Снаряд летить у напрямку гравця	Снаряд летить вниз до гравця	Pass
22	Поява Boss1	Набрати 5000 очок	Boss1 з'являється зверху	Boss1 з'явився	Pass
23	Стрільба Boss1	Спостерігати за Boss1	Стріляє 3 снарядами одночасно	Стріляє 3 снарядами	Pass
24	Поява Boss2	Набрати 10000 очок	Boss2 з'являється зверху	Boss2 з'явився	Pass
25	Поява Boss3	Набрати 15000 очок	Boss3 з'являється зверху	Boss3 з'явився	Pass
26	Анімація вибуху	Знищити ворога	З'являється анімація вибуху	Анімація відтворилась	Pass
27	Зміна фону	Набрати 3000 очок	Фон змінюється на новий	Фон змінився	Pass
28	Звуки пострілів	Натиснути на «Space»	Звук пострілів	Звук чутно	Pass
29	Звук у меню	Звук у меню, початку гри	Звук музики у меню	Звук чутно	Pass
30	Звук у битві	Звук у процесі гри	Звук та заміна на іншу музику	Музика змінюється та	Pass
31	Game over	Втратити все здоров'я	Показує екран завершення	Екран з'явився	Pass
32	Відображення рахунку	Влетіти і стріляти у ворогів	Рахунок збільшується	Збільшується коректно	Pass
33	Рекорд зберігається	Встановити рекорд	Hi-score залишається	Залишається, не змінюється	Pass
34	Вийти за допомогою клавіші	Натиснути клавішу «Enter» на «Exit»	Вихід з гри	Вихід з програмного забезпечення	Pass

Метод «білої скриньки»

Розроблено **457 модульних тестів**

438 тестів задовільні (Passed)

Виявлено **19 дефектів**:
Дефекти граничних умов видалення об'єктів
Недетермінована поведінка ворогів біля стін

```
========== short test summary info ==========
FAILED tests/test_bosses.py::TestBossComparison::test_boss1_unique_teleport_attribute - AttributeError: type object 'Boss1' has no attribute 'get_rect'
FAILED tests/test_bullets.py::TestBulletInitialState::test_bullet_initial_y_near_player - assert 690 == 700
FAILED tests/test_bullets.py::TestBulletBoundaryValues::test_bullet_not_removed_at_top_edge - assert 0 == 1
FAILED tests/test_bullets.py::TestBulletBoundaryValues::test_bullet_not_removed_one_pixel_above_top - assert 0 == 1
FAILED tests/test_bullets.py::TestBulletBoundaryValues::test_bullet_removed_when_below_bottom_edge - assert 1 == 0
FAILED tests/test_bullets.py::TestBulletBoundaryValues::test_bullet_removed_exactly_at_height_plus_one - assert 1 == 0
FAILED tests/test_bullets.py::test_bullet_removal_boundary[801-False] - assert (1 == 1) == False
FAILED tests/test_bullets.py::test_bullet_removal_boundary[900-False] - assert (1 == 1) == False
FAILED tests/test_enemies.py::TestEnemyMovement::test_enemy1_bounces_off_top_wall - assert 0 > 0
FAILED tests/test_enemies.py::TestEnemyMovement::test_enemy1_bounces_off_bottom_wall - assert 0 < 0
FAILED tests/test_meteors_refill.py::TestMeteorsBoundaryRemoval::test_meteor_not_removed_one_below_bottom_threshold - assert 0 == 1
FAILED tests/test_meteors_refill.py::TestMeteorsBoundaryRemoval::test_meteor_not_removed_one_below_right_threshold - assert 0 == 1
FAILED tests/test_meteors_refill.py::TestMeteors2BoundaryRemoval::test_meteor2_not_removed_one_below_threshold - assert 0 == 1
FAILED tests/test_meteors_refill.py::TestBlackHoleBoundaryRemoval::test_blackhole_not_removed_one_below_threshold - assert 0 == 1
FAILED tests/test_meteors_refill.py::TestExtraScoreBoundaryRemoval::test_extra_score_not_removed_one_below_threshold - assert 0 == 1
FAILED tests/test_meteors_refill.py::test_meteors_bottom_boundary[869-True] - assert (0 == 1) == True
FAILED tests/test_meteors_refill.py::test_meteors_right_boundary[1249-True] - assert (0 == 1) == True
FAILED tests/test_meteors_refill.py::test_meteors2_bottom_boundary[1099-True] - assert (0 == 1) == True
FAILED tests/test_meteors_refill.py::test_extra_score_bottom_boundary[899-True] - assert (0 == 1) == True
========== 19 failed, 438 passed in 1.28s ==========
```

Покриття коду: **93%**

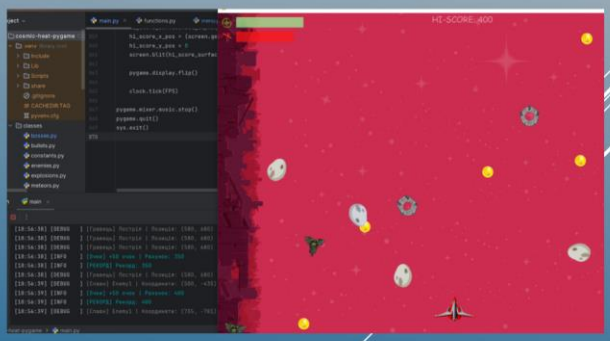
```
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
classes\bosses.py           224     21    91%   90, 129, 167, 190, 197-208, 302-305, 207-210, 212-215, 229
classes\bullets.py           16      0   100%
classes\constants.py          8      0   100%
classes\enemies.py          109     15    86%   87-106
classes\explosions.py        60      0   100%
classes\meteors.py           67      3    96%   33, 59, 88
classes\player.py            57      0   100%
classes\refill.py            94      4    96%   33, 62, 91, 116
-------------------------------------------------------
TOTAL                       635     43    93%
```

Метод «сірої скриньки»

Інтеграційне тестування: 13 тест-кейсів, усі задовільні (**Pass**)

Система логування підтвердила коректну взаємодію між модулями

Тестування продуктивності (FPS-оверлей): Середній FPS: **60 кадрів/с**



Аналіз виявлених дефектів

Всього виявлено **20 дефектів:**

1 – середньої критичності (шкали ресурсів), «чорна скринька»
19 – низької критичності (граничні умови), «біла скринька»
0 – «сіра скринька»

Жодного критичного дефекту не виявлено

Оцінка якості за ISO/IEC 25010

Характеристика	Оцінка
Функціональна придатність	Висока
Надійність	Висока
Продуктивність	Висока
Підтримуваність	Висока
Сумісність	Достатня
Гнучкість	Достатня
Взаємодія	Середня
Безпека	Не застосовується

Висновки

Застосування комплексного підходу до тестування програмного забезпечення на основі методів «чорної», «білої» та «сірої» скриньки дозволило провести повноцінну перевірку прикладного програмного забезпечення «Cosmic Heat» на всіх рівнях, саме – від зовнішньої функціональності до внутрішньої логіки та інтеграції між модулями.

Отримані результати підтвердили що жоден із методів окремо не забезпечує повноцінної оцінки якості – лише їх послідовне та взаємодоповнююче застосування дає об'єктивну та цілісну картину стану програмного продукту.

ДЯКУЮ ЗА УВАГУ!