Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти бакалавра

за спеціальності 121 – Інженерія програмного забезпечення

На тему: Порівняльний аналіз методологій тестування програмних продуктів на різних платформах

Засвідчую, що в цій кваліфікаційній роботі немає
запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ-20-2
_____/І.І Гречка/

Керівник
кваліфікаційної роботи _____ /Н.Н.Шаповалова/

Завідувач кафедри _____ /А.М.Стрюк/

Кривий Ріг
2024

Криворізький національний університет

Факультет: Інформаційних технологій

Кафедра: Моделювання та програмного забезпечення

Ступінь вищої освіти: бакалавр

Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри

_____ А.М.Стрюк

«__» _____ 20__р.

ЗАВДАННЯ

на кваліфікаційну роботу

студенту групи ІПЗ-20-2 Гречка Іван Ігорович

1. Тема: Порівняльний аналіз методологій тестування програмних продуктів на різних платформах. Затверджено наказом по КНУ №__ від «__» _____ 2024р.
2. Термін подання студентом закінченої роботи : «_» _____ 2024р.
3. Вихідні дані по роботі: розроблювана система повинна мати порівняльний аналіз систем тестування програмного забезпечення
4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): провести порівняльний аналіз ринку, визначитися з вимогами до фінальної версії розробленого програмного забезпечення, спроектувати систему тестування програмного забезпечення, її функціонал, розробити проектовану систему та протестувати її, перевірити коректну роботу усього функціоналу.
5. Перелік ілюстративного матеріалу: скріншоти екранних форм, блок-схеми функціоналу системи.

РЕФЕРАТ

ТЕСТУВАННЯ, QA, МЕТОДОЛОГІЇ ТЕСТУВАННЯ

Пояснювальна записка: 60 сторінок, 11 зображень, 6 таблиць, 7 джерел, 1 додаток.

Робота розглядає актуальність тестування програмного забезпечення в контексті сучасного світу, де ці технології стають все більш важливими в усіх сферах життя. Зокрема, зазначається, що зростання складності програмних продуктів призводить до потреби у розвитку ефективних методологій тестування, які б враховували різноманітність платформ, на яких вони використовуються.

Особлива увага приділяється мобільним додаткам, веб-сервісам та пристроям IoT, які зростають у популярності. Це призводить до необхідності тестування програм на різних платформах, що охоплюють різні операційні системи та середовища.

Порівняльний аналіз методологій тестування допомагає виявити найбільш ефективні підходи для конкретних умов та вимог. Також зазначається, що швидкий темп розвитку технологій та випуску нових версій програмного забезпечення підкреслює потребу у постійному оновленні та удосконаленні методів тестування. Це важливо для забезпечення високої якості та надійності програмних продуктів у контексті швидко змінюючихся технологічних умов. Такі дослідження відкривають шлях до постійного удосконалення процесів розробки та тестування програмного забезпечення на різних платформах.

ABSTRACT

TESTING, QA, TESTING METHODOLOGIES

Explanatory note: 60 pages, 12 images, 6 tables, 7 sources, 1 appendix.

The paper considers the relevance of software testing in the context of the modern world, where these technologies are becoming increasingly important in all spheres of life. In particular, it is noted that the increasing complexity of software products leads to the need to develop effective testing methodologies that would take into account the variety of platforms on which they are used.

Particular attention is paid to mobile applications, web services, and IoT devices, which are growing in popularity. This leads to the need to test applications on different platforms covering different operating systems and environments.

A comparative analysis of testing methodologies helps to identify the most effective approaches for specific conditions and requirements. It is also noted that the rapid pace of technology development and the release of new software versions emphasizes the need to constantly update and improve testing methods. This is important to ensure high quality and reliability of software products in the context of rapidly changing technological conditions. Such research paves the way for continuous improvement of software development and testing processes on various platforms.

ЗМІСТ

ВСТУП

В сучасному світі, де інформаційні технології відіграють важливу роль у всіх сферах діяльності, тестування програмного забезпечення стає необхідною складовою для багатьох аспектів нашого життя, від бізнесу та медицини до освіти та розваг. Проте зростання складності програмних продуктів породжує виклики у їх якісному забезпеченні, що в свою чергу підкреслює актуальність дослідження методологій тестування програмного забезпечення на різних платформах.

Зростаюча популярність мобільних та веб-додатків, а також розширення IoT (Internet of Things) приводять до необхідності тестування програм на різних платформах, таких як різні операційні системи, пристрої та середовища. Порівняльний аналіз методологій тестування на різних платформах дозволяє зрозуміти, які підходи є ефективними та найбільш підходящими для конкретних умов і вимог, а також сприяє розробці кращих стратегій тестування.

Крім того, з урахуванням швидкого темпу розвитку технологій і випуску нових версій програмного забезпечення, актуальність постійно переоцінювати і підлаштовувати методології тестування на різних платформах стає надзвичайно важливою для забезпечення високої якості та надійності програмних продуктів. Тому дослідження у цій області відкриває шлях до постійного вдосконалення і удосконалення процесів розробки та тестування програмного забезпечення на різних платформах.

У роботі буде проведений аналіз існуючих методів тестування та впровадження їх у програмний продукт Також буде розглянуто практичні аспекти реалізації різних методів тестування та їх можливостей у різних сферах людських діяльностей.

1 СУЧАСНИЙ СТАН І АКТУАЛЬНІСТЬ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1.1 Актуальність теми кваліфікаційної роботи

Актуальність теми кваліфікаційної роботи полягає в тому, що в сучасному інформаційному світі розробники програмного забезпечення постійно стикаються з необхідністю вибору оптимальних методологій для тестування своїх продуктів на різних платформах. З появою нових технологій, фреймворків та платформ для розробки програмного забезпечення, стає важливим зрозуміти, які методології тестування найбільш ефективні для кожного конкретного випадку.

Аналіз методологій тестування допомагає розробникам обирати найбільш підходящі під їхні потреби і умови проекту. Додатково, у зв'язку з швидким розвитком індустрії програмного забезпечення, постійно виникають нові методології тестування, і важливо слідкувати за цими тенденціями та оцінювати їх ефективність.

Таким чином, ми маємо велике практичне значення для розробників програмного забезпечення та інженерів з якістю, сприяючи покращенню процесів розробки та впровадження програмного забезпечення.

1.2 Цілі та завдання кваліфікаційної роботи

Ціллю кваліфікаційної роботи є проведення комплексного порівняльного аналізу методологій тестування програмних продуктів на різних платформах з метою визначення їхньої ефективності та придатності для різних типів проектів. Для досягнення цієї загальної мети передбачаються наступні завдання:

1. Зібрати та систематизувати інформацію: Провести огляд літературних джерел, наукових статей та онлайн-ресурсів для збору інформації про різні методології тестування,, а також їхні переваги та недоліки.

2. Вибір методологій та платформ для порівняння: Вибрати найбільш популярні та використовувані методології тестування (наприклад, BDD, TDD,DDD) та платформи (веб, мобільні, десктопні), які будуть порівнюватися у роботі.
3. Розробка критеріїв оцінки: Розробити об'єктивні критерії для оцінки ефективності та придатності кожної методології тестуванн, враховуючи особливості проектів.
4. Проведення експериментів та аналіз результатів: Провести серію експериментів з застосування обраних методологій тестування на різних платформах та проаналізувати отримані результати згідно з розробленими критеріями оцінки.
5. Формулювання висновків та рекомендацій: На основі проведеного аналізу сформулювати висновки щодо ефективності та придатності кожної методології тестування на різних платформах та надати рекомендації для вибору оптимальної стратегії тестування для конкретних типів проектів.

1.3 Аналіз вимог

Під час розгляду різноманітних джерел з теми порівняльного аналізу методологій тестування програмних продуктів на різних платформах, було виявлено кілька ключових аспектів та проблем, з якими можна зіткнутися у цьому контексті.

Однією з основних проблем є неоднаковість точності та надійності функцій, що використовуються для тестування програмного забезпечення на різних платформах. Наприклад, деякі методи тестування можуть бути ефективнішими на одній платформі, але непридатними для іншої.

Другою проблемою є обмежені можливості адаптації методологій тестування до різних платформ. Одна й та ж методологія може давати різні результати на різних платформах через їхні специфічні особливості.

Додатково, слід враховувати проблеми, пов'язані з ефективністю та

швидкістю виконання тестів на різних платформах. Деякі методи тестування можуть вимагати значних обчислювальних ресурсів або тривалого часу для виконання на певних платформах, що ускладнює їхнє використання.

Отже, для успішного порівняльного аналізу методологій тестування програмних продуктів на різних платформах важливо розглянути ці проблеми та знайти оптимальні рішення для їх вирішення. Аналіз вимог можно поділити на такі частини:

Функціональні вимоги:

1. Вибір методології тестування: Система повинна надати можливість користувачам обирати між різними методологіями тестування, такими як BDD, TDD, або іншими, в залежності від їх потреб.
2. Вибір платформи для тестування: Користувач повинен мати можливість вибирати платформу для тестування, таку як веб-додатки, мобільні додатки або десктопні програми, залежно від специфіки їхнього проекту.
3. Запуск тестів: Система має надавати можливість автоматичного запуску тестів для вибраної методології тестування та платформи.
4. Відображення результатів тестів: Після завершення тестування система повинна показувати результати тестів, включаючи інформацію про пройдені, провалені та пропущені тести.

Нефункціональні вимоги:

1. Швидкість тестування: Система повинна бути здатна виконувати тести швидко та ефективно, незалежно від обсягу та складності тестів.
2. Масштабованість: Система має бути готовою для виконання тестів на різних масштабах, від маленьких проектів до великих корпоративних систем.
3. Надійність: Система повинна бути стабільною та надійною під час виконання тестів, щоб забезпечити точні та надійні результати.
4. Підтримка платформ: Система має підтримувати різні платформи для тестування, забезпечуючи однакову ефективність на кожній з них.

Рисунок 1.1 – Приклад планування вимог перед тестуванням

Це допоможе забезпечити розширення можливостей та забезпечити ефективне тестування програмних продуктів на різних платформах, але варто

Для реалізації тестування за методологією BDD (Behavior-driven development) на мові програмування Python можна використовувати такі бібліотеки:

1. **Pytest-BDD** з Python-Gherkin: Ця бібліотека використовує синтаксис Gherkin для написання тестових сценаріїв, що описують поведінку програми з точки зору користувача. Pytest-BDD інтегрується з Pytest, що дозволяє виконувати тестування в середовищі Python. Вона дозволяє писати тести на основі ключових слов Gherkin, таких як Given, When, Then, що робить їх зрозумілими для всіх учасників проекту.
2. **Cucumber**: Це інструмент для виконання тестів на основі Gherkin-синтаксису, який початково був розроблений для мови програмування Ruby, але також має підтримку для Python. За допомогою Cucumber, можна писати та виконувати тестові сценарії на мові Gherkin для тестування програмного забезпечення. Використання

Cucumber у сполученні з Python дозволяє легко і ефективно виконувати тестування за методологією BDD.

Для реалізації тестування за методологією TDD (Test Driven Development) на мові програмування Python можна використовувати такі бібліотеки:

1. **unittest**: Вбудований модуль в Python, який надає фреймворк для тестування на основі юніт-тестів. Він дозволяє писати тести та перевіряти правильність роботи програмного коду на основі різних входів та очікуваних результатів.
2. **pytest**: Фреймворк для тестування Python, який надає зручний синтаксис для написання тестів та автоматизованого виконання їх. Він має багато вбудованих функцій для підтримки різних видів тестів і сприяє розробці за методологією TDD.

2 МЕТОДОЛОГІЇ ТЕСТУВАННЯ ПРОГРАМНИХ ПРОДУКТІВ

За останні роки в галузі тестування програмного забезпечення стали популярними методології BDD,TDD,DDD, тому в основному будемо розглядати їх, але так же звернемо увагу і на інші методології.Ефективність використання методологій тестування полягає у забезпеченні високої якості програмного забезпечення, зменшенні часу та витрат на розробку, зниженні кількості помилок та підвищенні задоволення клієнтів. Ось деякі ключові аспекти ефективності методологій тестування:

1. Покращення якості продукту: Методології тестування допомагають виявляти помилки та дефекти на ранніх етапах розробки, що дозволяє їх виправляти швидше та зменшує ризик їх появи в продукційному середовищі.
2. Оптимізація витрат часу та ресурсів: Ефективні методології дозволяють автоматизувати тестування, що зменшує час, необхідний для виконання тестів, та витрати на їх проведення вручну.
3. Відповідність вимогам клієнта: Впровадження методологій тестування допомагає забезпечити, що продукт відповідає вимогам та очікуванням клієнта, оскільки тестування базується на специфікаціях та вимогах.
4. Підвищення впевненості в продукті: Послідовне та систематичне тестування допомагає підвищити впевненість в якості програмного забезпечення, як для команди розробників, так і для клієнтів.
5. Зменшення ризиків та витрат на підтримку: Попереднє тестування дозволяє виявляти та виправляти проблеми ще до випуску продукту в експлуатацію, що допомагає знизити ризики та витрати на подальше управління та підтримку.

2.1 Принципи та особливості TDD

Test-Driven Development (TDD), або розробка через тести, є підходом до розробки програмного забезпечення, де розробка нового функціоналу

починається з написання тестів перед написанням самого коду. Процес TDD базується на коротких ітераціях, де шкірна ітерація починається з написання тесту, що описує очікувану поведінку нової функціональності.

TDD використовує ітеративний підхід, відомий як "Red-Green-Refactor". Спочатку пишеться тест, який не проходить (червоний), потім пишеться мінімальний код, щоб зробити тест прохідним (зелений), а потім проводиться рефакторинг коду для покращення його структури та чистоти.

Основна особливість TDD – це швидкий зворотний зв'язок розробнику. Після написання тесту розробник відразу бачить, чи проходить код чи ні, що дозволяє швидко виправляти помилки. Використання TDD також сприяє створенню гнучкішої та добре структурованої архітектури програми.

Оскільки код пишеться під контролем тестів, це полегшує створення чистого та модульного коду. Таким чином, TDD сприяє покращенню якості програмного забезпечення, забезпечує швидкий зворотний зв'язок та підвищує довіру до коду за рахунок високого рівня покриття тестами та створення більш гнучкої архітектури.

2.1.1 Цикли TDD: "red-green-refactor"

Основні кроки розробки через TDD виглядають так:

1. Написання тесту: Розробник спочатку пише тест, який описує очікувану поведінку нової функції або модуля. Цей тест спочатку має провалитися, оскільки функціональність ще не реалізована.
2. Запуск тесту: Розробник запускає тест і переконується, що він дійсно провалюється (адже потрібною функціональності ще немає).
3. Написання коду: Розробник пише необхідний мінімальний код, щоб зробити тест успішним.
4. Запуск тестів: Після написання коду розробник запускає усі тести, включаючи новий, щоб переконатися, що новий код не порушує існуючої функціональності.

5. Рефакторинг: Після того як тести пройшли успішно, розробник може провести рефакторинг коду, покращуючи його якість без зміни зовнішньої поведінки.
6. Повторення: Цей цикл повторюється для кожної нової функціональності чи модуля, що додається до програми.

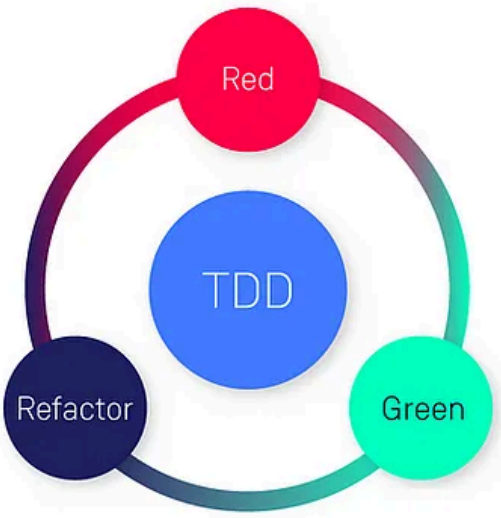

Рисунок 2.1.1 – Цикл тестування TDD

Переваги TDD:

- Раннє виявлення помилок: Написання тестів перед реалізацією коду дозволяє виявити помилки на ранніх етапах розробки, що спрощує їх виправлення і покращує якість коду.
- Документування: Тести служать як специфіка функціональності. Це дозволяє швидко зрозуміти, як використовувати та як має працювати код.
- Впевнені зміни: Можливість внесення змін у код з впевненістю, що вони не порушать наявну функціональність, оскільки будь-які зміни повинні відповідати існуючим тестам.
- Покриття коду тестами: TDD спонукає до створення ширшого покриття коду тестами, що сприяє підвищенню якості програми.

- Рефакторинг: Завдяки наявності набору тестів можна проводити рефакторинг коду з впевненістю, що функціональність залишиться незмінною.

Недоліки TDD:

- Час на вивчення: Вчитися писати тести перед кодом може зайняти час і зусилля. Розробники повинні навчитися добре структурувати свої тести.
- Додатковий час на написання коду: Початкове написання тестів може забрати більше часу, ніж просто написання коду. Однак це може зекономити час пізніше при усуненні помилок та рефакторингу.
- Необхідність підтримувати тестовий набір: Тестовий набір потрібно підтримувати разом з кодом. Якщо програма розвивається, тести можуть вимагати оновлення під новий функціонал.
- Не підходить для всіх видів проектів: Деякі види програм можуть бути складніше піддаються тестуванню за допомогою TDD, наприклад, програми з великою кількістю інтерфейсних елементів або важкою структурою.

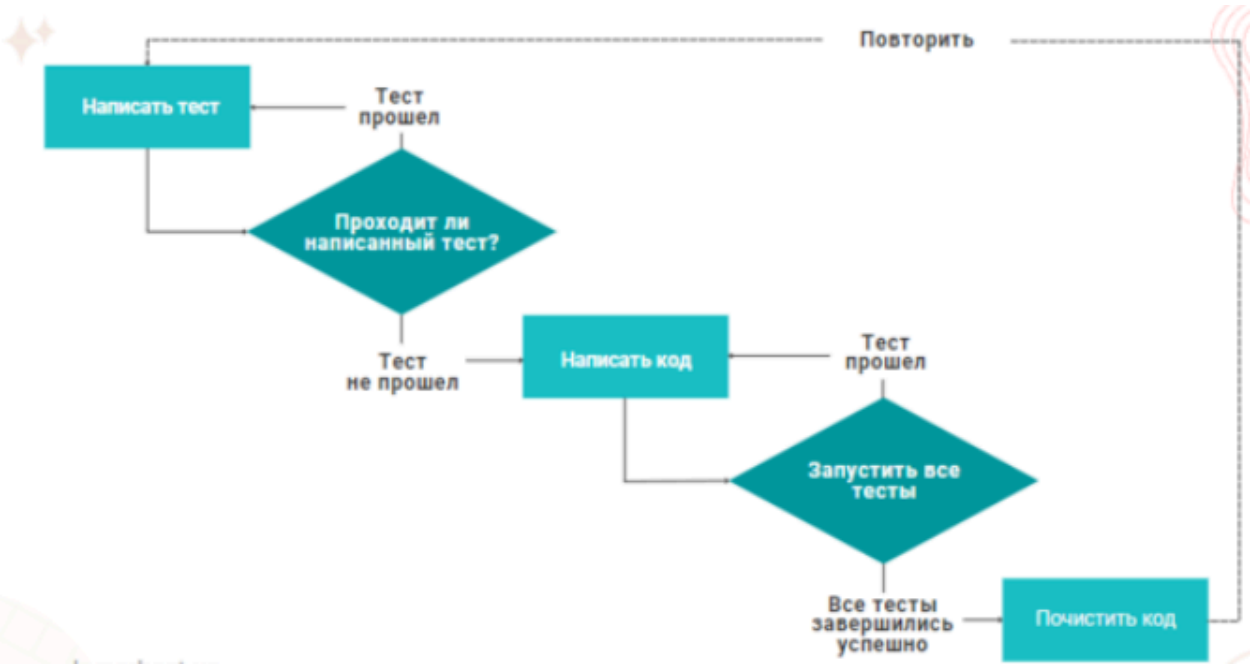

Рисунок 2.1.2 – Приклад схеми роботи з методологією TDD

2.1.2 Приклади використання TDD для розробки та тестування програмного забезпечення.

Давайте розглянемо приклад використання Test-Driven Development (TDD) для розробки веб-програми з використанням Python і фреймворку Flask.

Створимо файли для нашого проекту. Припустимо, ми будемо використовувати структуру проекту такого виду:

```
project/
│
├── app/
│   ├── __init__.py
│   ├── todo.py
│
└── tests/
    ├── __init__.py
    ├── test_todo.py
```

У файлі app/todo.py визначимо нашу програму Flask і простий ендпоінт для управління завданнями:

```
from flask import Flask, jsonify, request
app = Flask(__name__)
tasks = []

@app.route('/tasks', methods=['GET'])
def get_tasks():
    return jsonify(tasks)
@app.route('/tasks', methods=['POST'])
def add_task():
    data = request.get_json()
    if 'title' in data:
        title = data['title']
        tasks.append(title)
```

```
        return jsonify({'message': 'Task added successfully'}), 201
    else:
        return jsonify({'error': 'Title is required'}), 400
```

У файлі tests/test_todo.py напишемо тести для нашого веб-застосунку з використанням TDD:

```
import unittest
import json
from app import todo, app
class TestTodoApp(unittest.TestCase):
    def setUp(self):
        app.testing = True
        self.app = app.test_client()
    def test_get_tasks_empty(self):
        response = self.app.get('/tasks')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertEqual(data, [])
    def test_add_task(self):
        response = self.app.post('/tasks', json={'title': 'Buy groceries'})
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 201)
        self.assertEqual(data['message'], 'Task added successfully')
        response = self.app.get('/tasks')
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 200)
        self.assertIn('Buy groceries', data)
    def test_add_task_missing_title(self):
        response = self.app.post('/tasks', json={})
        data = json.loads(response.data.decode())
        self.assertEqual(response.status_code, 400)
        self.assertEqual(data['error'], 'Title is required')
if __name__ == '__main__':
    unittest.main()
```

Тепер ми можемо запустити наші тести для перевірки роботи нашого веб-додатку:

```
python -m unittest discover -s tests
```

Ця команда запустить усі тести з папки tests та покаже результати. Якщо всі тести проходять успішно, можна бути впевненим, що веб-додаток працює відповідно до специфікації.

У цьому прикладі ми використовували TDD для розробки веб-програми на основі Flask. Спочатку ми визначили тести для функціональності додавання та отримання завдань, а потім реалізували саму функціональність, щоб переконатися, що всі тести проходять успішно. Цей підхід дозволяє забезпечити надійність та відповідність додатку заданим вимогам.

2.2 Принципи та особливості BDD

Методологія BDD (Behavior-Driven Development), надає можливість бізнес-аналітикам, розробникам та тестувальникам спільно працювати над специфікацією та тестуванням продукту. У цьому контексті Gherkin та Pytest-BDD є двома часто використовуваними інструментами. Проте, існують й інші методології, які можуть бути застосовані для досягнення схожих цілей. Ця теза спрямована на порівняльний аналіз ефективності застосування Gherkin та Pytest-BDD у порівнянні з іншими методологіями в тестуванні програмного забезпечення. Порівняння буде здійснене з урахуванням таких аспектів, як зручність використання та швидкість розробки.

Окрема увага буде звернута на їхню здатність співпрацювати з іншими інструментами та технологіями, такими як фреймворки для автоматизованого тестування, системи управління версіями та інші інструменти, які широко використовуються у сучасних проектах розробки ПЗ.

Отримані результати дослідження допоможуть зрозуміти переваги та недоліки кожної з методологій в різних сценаріях застосування та

сприятимуть вибору найбільш підходящого підходу до тестування продуктів на різних платформах з урахуванням конкретних вимог та умов проекту.

2.2.1 Ключові концепції BDD: специфікації на основі поведінки.

При аналізі синтаксису методологій Gherkin та Pytest-BDD важливо врахувати їхню специфіку та особливості. Синтаксис Gherkin базується на спеціальному діалекті, який використовує ключові слова для опису поведінки системи. Основними ключовими словами є "Feature", "Scenario", "Given", "When", "Then" і "And", які допомагають структурувати тестові сценарії.

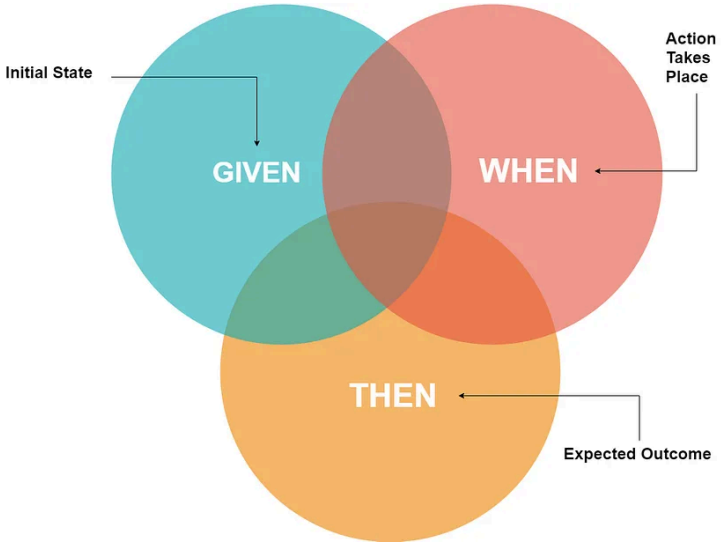

Рисунок 2.2.1 – Приклад схеми роботи з методологією TDD

Наприклад:

```
Feature: Покупка товару
Scenario: Додавання товару до кошика
  Given користувач має відкриту сторінку товару
  When користувач додав товар до кошика
  Then товар повинен бути в кошику
```

Синтаксис Pytest-BDD базується на стандартному синтаксисі фреймворку Pytest, але додає до нього можливості для опису тестових

сценаріїв у стилі BDD. Використання декораторів @given, @when, @then та @scenario дозволяє створювати тести у зрозумілій для бізнес-користувачів формі.

Наприклад:

```
from pytest_bdd import given, when, then, scenario
@scenario('purchase.feature', 'Додавання товару до кошика')
def test_adding_to_cart():
  pass
@given("користувач має відкриту сторінку товару")
def open_product_page():
  pass
@when("користувач додав товар до кошика")
def add_product_to_cart():
  pass
@then("товар повинен бути в кошику")
def verify_product_in_cart():
  pass
```

Переваги:

- Тестування з точки зору користувача.
- Легко читати.
- Любий може написати.
- Тести не залежать від мови програмування.
- Легко модифікувати.

Недоліки:

- Може вимагати більше часу, ніж TDD.
- Вимагає залучення спеціалістів на етапі вимог.

2.2.2 Приклади використання BDD у програмному забезпеченні

Давайте рассмотрим пример использования BDD(Behiven Driven Development) для разработки веб-приложения с использованием Python и фреймворка Flask с Behave.

Создадим файл features/todo.feature для описания спецификаций приложения в формате Gherkin:

```
Feature: Managing tasks
 As a user
 I want to manage my tasks
 So that I can keep track of them
 Scenario: Viewing tasks
   Given there are no tasks
   When I view my tasks
   Then I should see an empty list of tasks
 Scenario: Adding a new task
   When I add a new task with title "Buy groceries"
   Then I should see the task "Buy groceries" in my task list
```

Створимо файл features/steps/todo_steps.py, де будуть описані кроки для кожного сценарію:

```
from behave import given, when, then
from app import app


@given('there are no tasks')
def step_impl(context):
   context.app = app.test_client()
   # Проверяем, что список задач пуст
   response = context.app.get('/tasks')
   assert response.status_code == 200
   assert len(response.json) == 0
```

```
@when('I view my tasks')
def step_impl(context):
    context.response = context.app.get('/tasks')
@then('I should see an empty list of tasks')
def step_impl(context):
    assert context.response.status_code == 200
    assert context.response.json == []
@when('I add a new task with title "{task_title}"')
def step_impl(context, task_title):
    context.task_title = task_title
    context.response = context.app.post('/tasks', json={'title': task_title})
@then('I should see the task "{task_title}" in my task list')
def step_impl(context, task_title):
    response = context.app.get('/tasks')
    assert context.response.status_code == 201
    assert task_title in response.json
```

Тепер ми можемо запустити тести BDD сценаріїв за допомогою behave.

Behave автоматично знайде та виконає всі фічі та кроки, описані у наших фіч-файлах та відповідних крокових визначеннях. Він буде використовувати програму Flask як клієнт API для взаємодії з нашим веб-додатком та перевірки результатів.

Цей приклад демонструє використання BDD для опису та перевірки поведінки веб-застосунків за допомогою специфікацій у стилі Gherkin та відповідних крокових визначень. BDD допомагає покращити розуміння вимог та поведінки програми на основі бізнес-кейсів, що робить тести більш читаними та доступними для учасників команди розробки.

3 ПОРІВНЯЛЬНИЙ АНАЛІЗ МЕТОДОЛОГІЙ ТЕСТУВАННЯ

3.1 Програмна реалізація тестувального середовища для методології BDD

Розробка гри - це складний і багатоетапний процес, в якому важливе місце посідає тестування. В даній роботі розглядається розробка та тестування гри з використанням методології BDD (Behavior-Driven Development), яка дозволяє уникнути багатьох проблем, що виникають на етапі розробки та тестування гри.

Потреби тестування гри:

1. Функціональність гри:
    - Перевірка коректності роботи основних ігрових механік, таких як переміщення персонажів, взаємодія з об'єктами, баланс гри тощо.
    - Тестування гри на різних рівнях складності для забезпечення задоволення гравців різного рівня навичок.
2. Графіка та анімація:
    - Перевірка відображення графіки та анімацій на різних пристроях та роздільних здатностях екрану.
    - Тестування оптимізації гри для плавної роботи на різних платформах.
3. Взаємодія з користувачем:
    - Перевірка зручності та інтуїтивності інтерфейсу гри.
    - Тестування реакції гри на введення гравця через різні пристрої вводу (клавіатура, миша, тачскрін тощо).
4. Стабільність та надійність:
    - Проведення тестів на виявлення помилок, що можуть спричинити збої гри або втрату даних.
    - Тестування гри на витривалість та здатність працювати під високим навантаженням.
5. Сценарії та ігровий процес:

- Створення сценаріїв гри для перевірки різних можливих варіантів геймплею.
- Тестування ігрового процесу на наявність можливості досягнення цілей гравця та вирішення завдань.

Визначення потреб тестування це важлиий етап який дозволяє перевірити роботу різних компонентів та функціоналу гри перед її випуском. Використання методології BDD дозволяє покращити якість тестування та спростити спілкування між розробниками та тестувальниками, що сприяє створенню якісної та стабільної гри для користувачів.

3.1.1 Розробка тестових сценаріїв

Розробка тестових сценаріїв для гри - це процес створення послідовностей дій, які тестувальники будуть виконувати для перевірки функціональності, якості та коректності роботи гри. Це важлива частина процесу тестування, оскільки допомагає забезпечити, що гра працює відповідно до очікуваних стандартів.

Навіщо потрібно:

1. Забезпечення якості продукту: Тестові сценарії дозволяють виявити потенційні проблеми та дефекти в роботі гри, що допомагає вирішити їх перед випуском гри на ринок.
2. Визначення охоплення тестування: Розробка широкого спектру тестових сценаріїв дозволяє покрити різні аспекти гри, від головних ігрових механік до взаємодії з інтерфейсом користувача.
3. Сприяння ефективності тестування: Чітко визначені сценарії дозволяють тестувальникам проводити тестування систематично та ефективно, зменшуючи час, необхідний для проведення тестів.
4. Забезпечення відповідності вимогам: Тестові сценарії можуть бути розроблені на основі вимог до гри, що допомагає переконатися, що розроблений продукт відповідає специфікаціям.

Розробка тестових сценаріїв є важливою частиною процесу тестування гри і дозволяє забезпечити високу якість та надійність продукту перед його випуском.

Наша гра розділяється на такі тестові сценарії:

1. Board
2. Combo-Merge
3. Combo-Merge-Reward
4. Destructing
5. Economic
6. Merge
7. Probability

В грі представлено кілька основних особливостей

1. Економіка монет:
    - Гравець може заробляти монети шляхом виконання різних завдань або досягнень у грі.
    - Монети використовуються для придбання нових будинків, покращень або інших корисних ресурсів.
2. Механіка знищення будинків:
    - Гравець може знищити будинок на ігровому полі, щоб звільнити місце для нових будинків або оптимізувати стратегію гри.
3. Механіка підвищення рівня будиночків:
    - Гравець може підвищувати рівень будинків шляхом їх злиття. Наприклад, три будинки одного рівня можуть злитися в один будинок більшого рівня.
4. Механіка переміщення будиночків:
    - Гравець може переміщати будинки по ігровому полю, щоб оптимізувати їх розміщення та створювати умови для створення комбо.
5. Турнірний будинок:

- Особливий тип будинку, який може з'являтися у визначених умовах або в режимі турнірів.
- Турнірні будинки можуть мати унікальні властивості або призначення, які додають грі варіативності та виклику.

6. Останній будинок:
    - Спеціальний елемент гри, який з'являється в кінці раунду або ігрової сесії.
    - Останній будинок може мати особливі властивості або наслідки, які впливають на результат гри або надають додаткові вигоди гравцю.

Таблиця 3.1.1 – Реалізація тестових сценаріїв для Board

Scenario Description	**Given**	**When**	**Then**
User puts the house at empty cell	Game board with default preset houses boards/give/simple_board/positive/default.yml	User have in board queue [1,1,1,1]	A new boards/then/simple_board/positive/result_board.yml should be generated by user as a result of the installation of the house at empty cell
User puts the house at used cell	Game board with default preset houses boards/give/simple_board/negative/default.yml	User have in board queue [1,1,1,1]	Nothing will happen

Checking the end of the game after the entire game board is full	Fully filled board with houses of different levels		Game is stopped
The user is trying to change the position of the already placed house	Random numbers of houses on the game table	The user tries to drag the house to another position	Nothing will happen

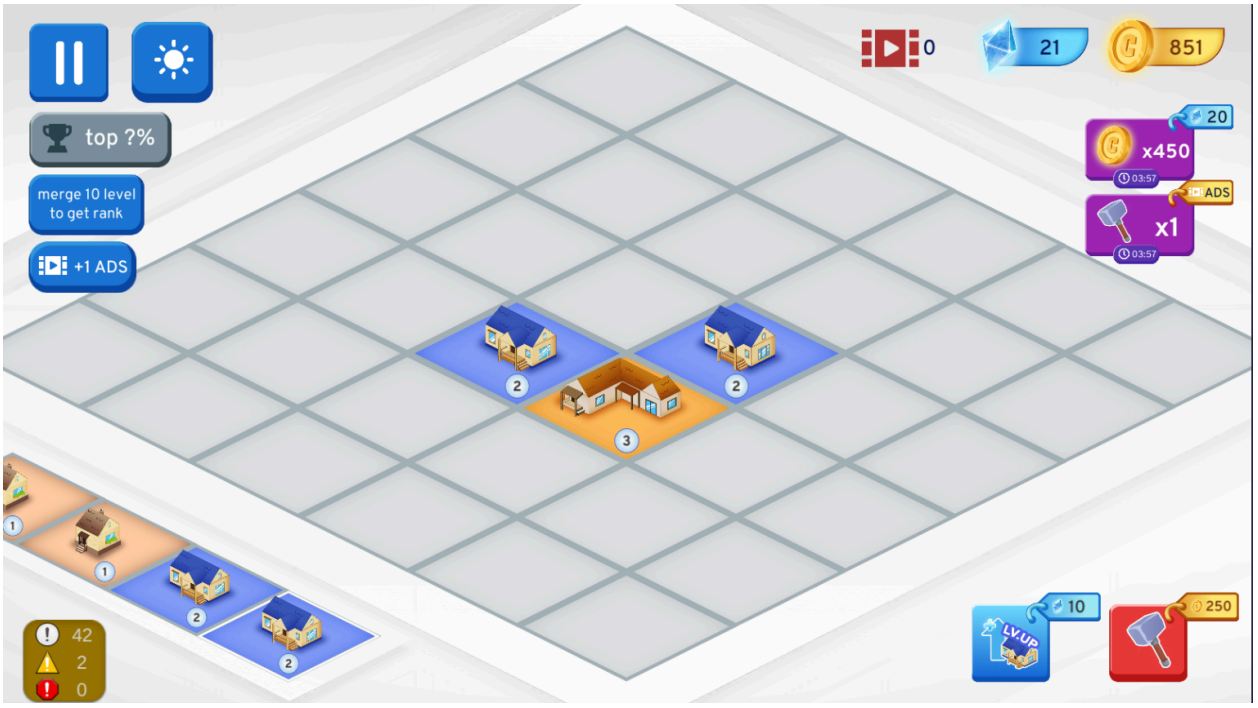

Рисунок 3.1.1 – Приклад дошки

Таблиця 3.1.2 – Реалізація тестових сценаріїв для Combo-Merge

Сценарій	**Given**	**When**	**Then**
Combo-merge function	Game board with default preset houses boards/give/combo_merge_board/positive/default.yml	User have in board queue [1,1,1,1]	From house lvl 1 should be generated a new boards/give/combo_merge_board/positive/result_board.yml with combo-merge result lvl 3 house User gets a crystal
Negative test to check the impossibility of combo-merging houses of different lvl	Game board with default preset houses boards/give/combo_merge_board/negative/default.yml	User have in board queue [1,1,1,3]	From house lvl 3 should not be generated a new boards/give/combo_merge_board/negative/result_board.yml with combo-merge result lvl 4 house

Ця таблиця представляє тестові сценарії для функції Combo-Merge і включає в себе дані перед умовою (Given), подію (When) та очікуваний результат (Then). Кожен сценарій розділений на різні кроки для зручності та чіткості.

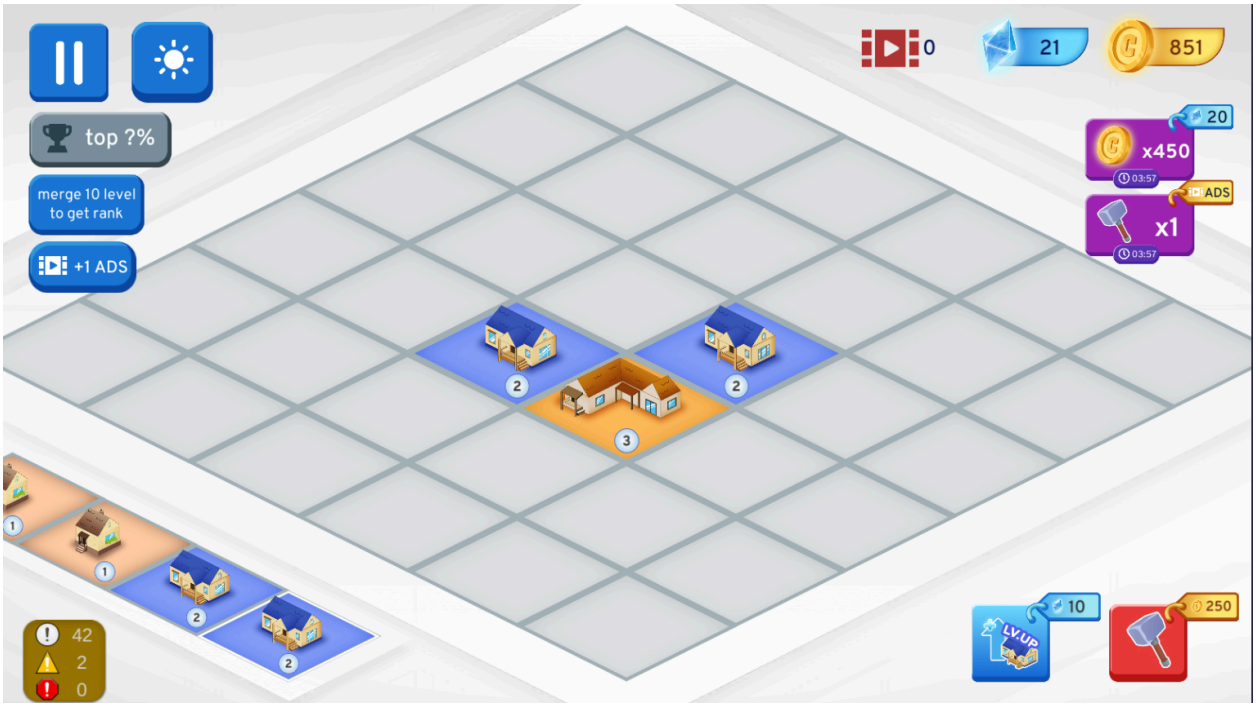

Рисунок 3.1.2 – Приклад функції з'єднання будинків

Таблиця 3.1.3 – Реалізація тестових сценаріїв для Destructing

Сценарій	Given	When	Then
Using the destruction function at home	Random numbers of houses on the game table	One possibility to use the destruction function	User destroys a random house
Using the destruction function at empty cell	Random numbers of houses on the game table	One possibility to use the destruction function	User try to destroy an empty cell

Getting the destroy function after receiving coins	Random numbers of houses on the game table	One possibility to use the destruction function	User gets 200 coins
Canceling the destroy function after clicking on it	Random numbers of houses on the game table	One possibility to use the destruction function	User clicks on destroy button
Getting the destroy function after starting a game		User starts a game	User gets one destroy function
Using the destruction function at spawn place	Random numbers of houses on the game table	One possibility to use the destruction function	Spawn board with random lvl of houses
Trying to destroy a house with 0 coins and no destruction function	Random numbers of houses on the game table		User tries to destroy a random house

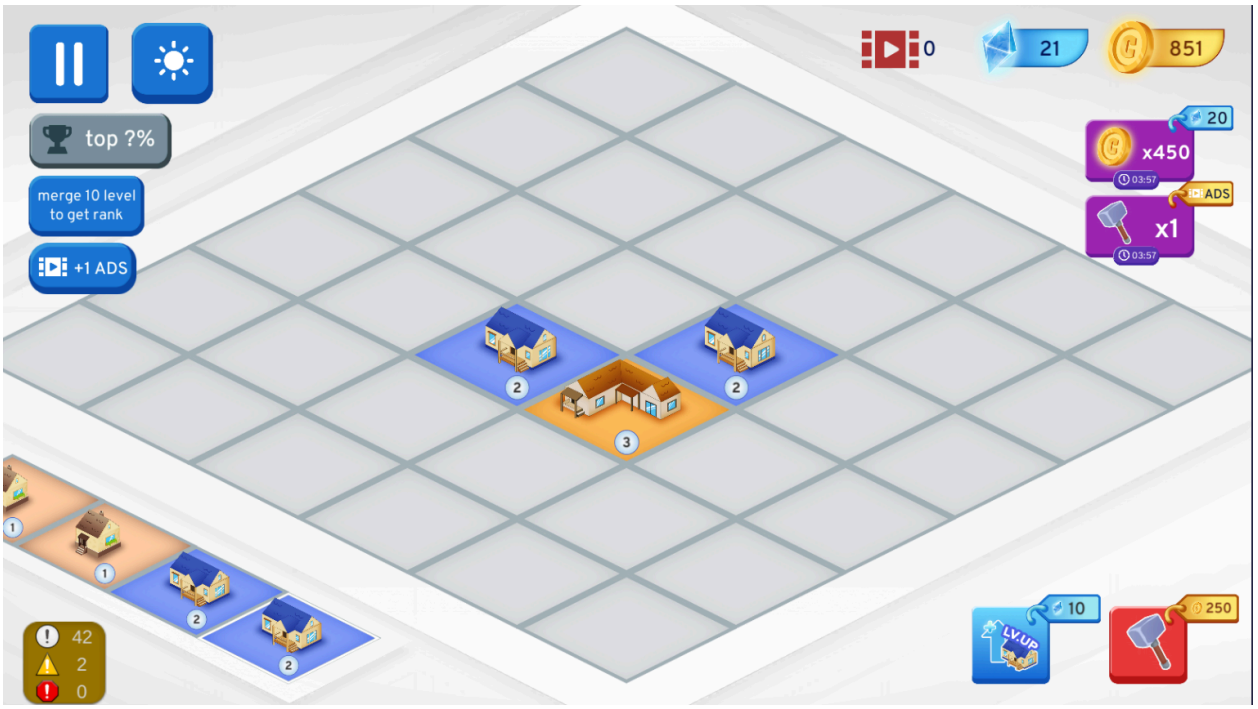

Рисунок 3.1.3 – Приклад функції Знищення

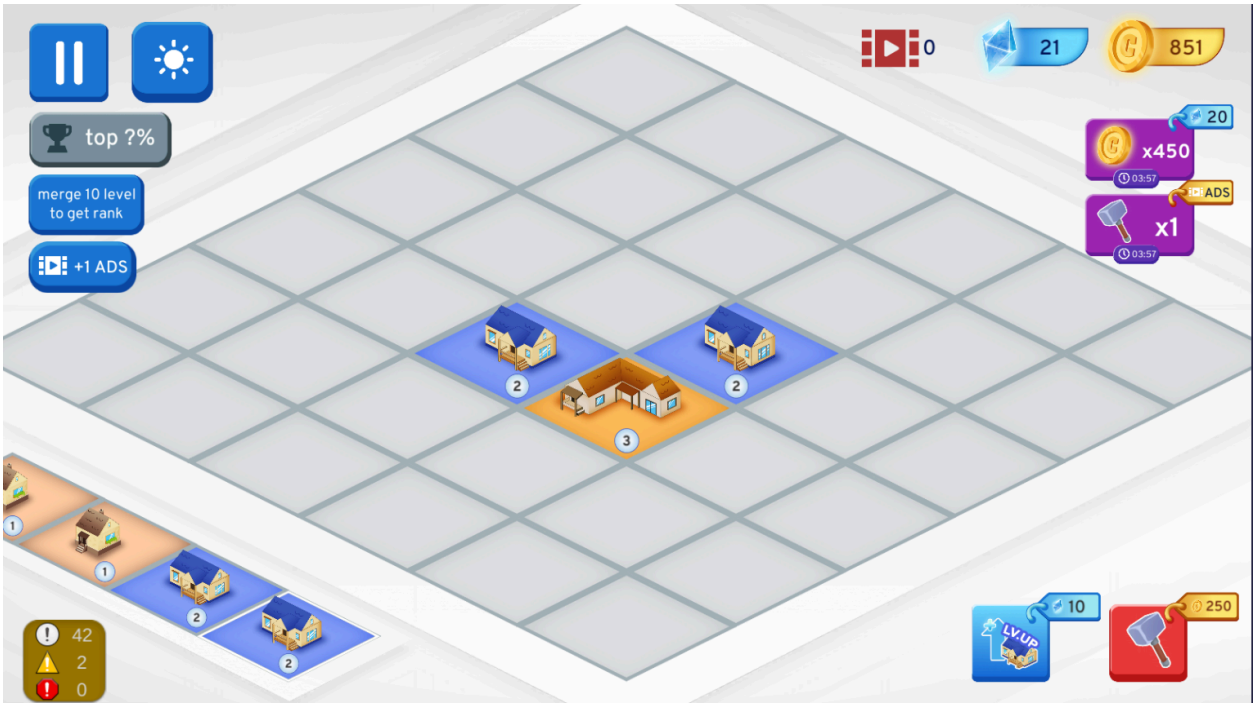

Рисунок 3.1.4 – Приклад активної функції Знищення

Таблиця 3.1.4 – Реалізація тестових сценаріїв для Economic

Сценарій	Given	When	Then
Getting n coins for placing a house	Random numbers of houses on the game table	User puts the house of random lvl on cell	User gets n coins
Getting n*20 coins for destroying a house	Random numbers of houses on the game table	User destroys a random house	User gets n*20 coins
Getting n+1 coins for using crystal	Crystal	User use crystal on a house	User gets n+1 coins
Getting n*x coins for merging x houses of n lvl	Random numbers of houses on the game table	User merges x houses of n lvl	User gets n*x coins

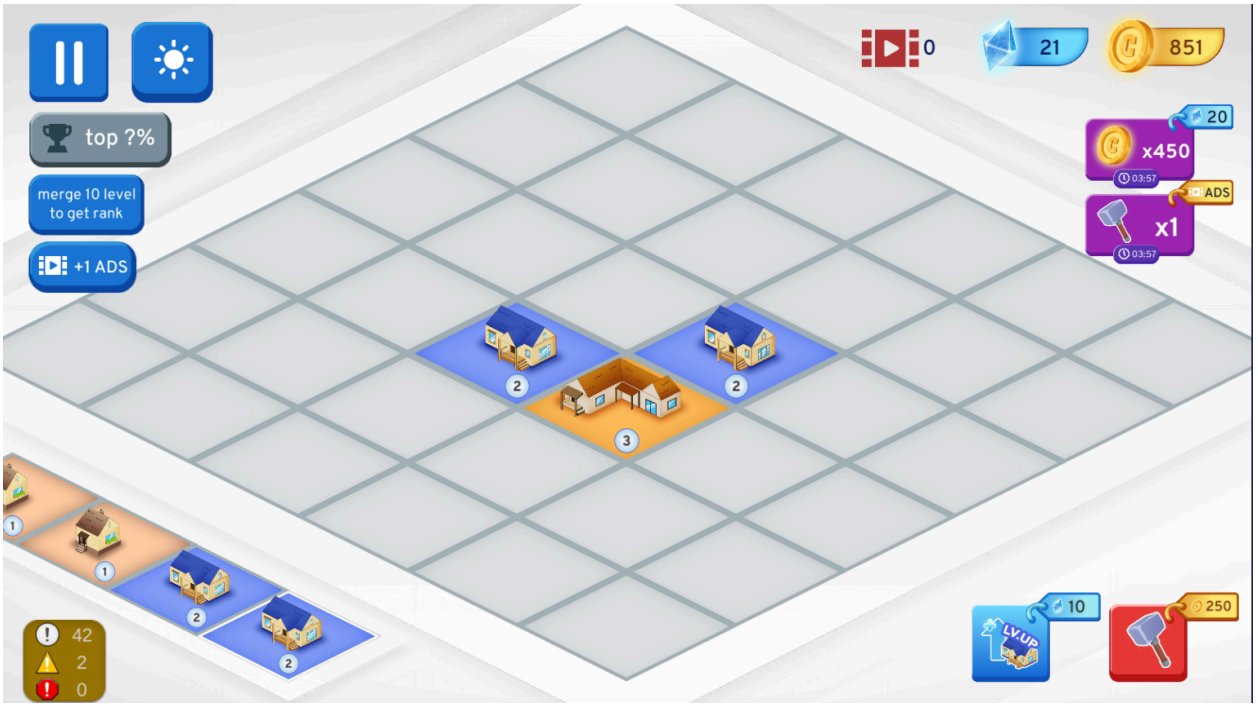

Рисунок 3.1.5 – Приклад функції Економіки

3.1.2 Написання основного тестового коду

Перед початком реалізації програмного коду нам потрібно розбити структуру проекту на логічну ієрархію.

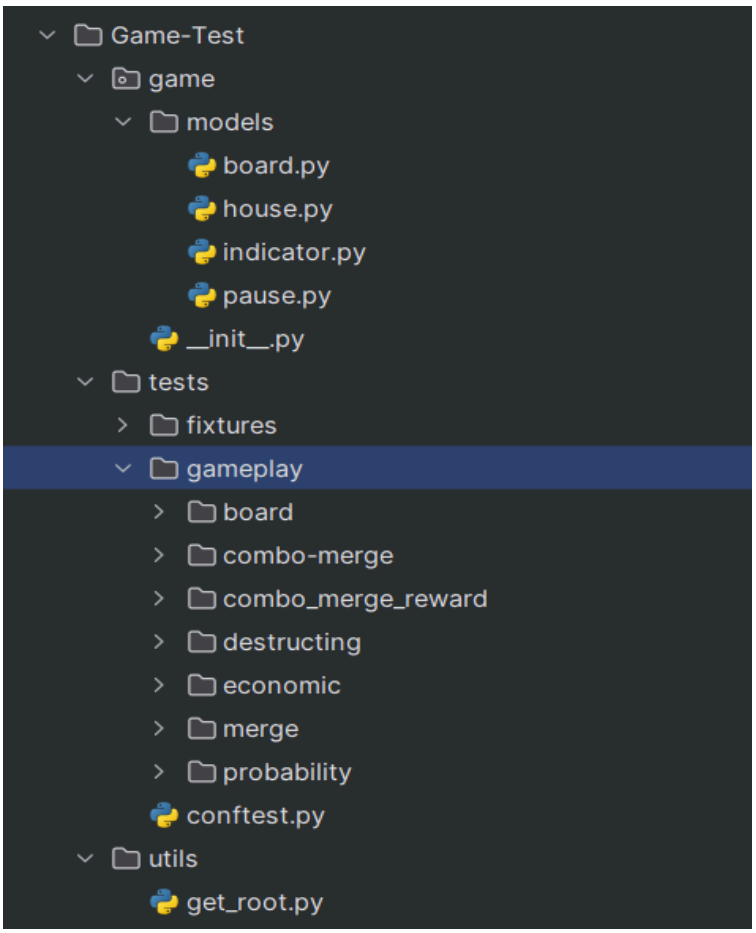

Рисунок 3.1.6 – Структура проекту

На рисунку 3.1.6 зображено приклад логічної ієрархії структури тестування програмного забезпечення.

1. game: Цей каталог містить модулі, пов'язані з грою.
    1.1. models: Модулі, які визначають структуру об'єктів у грі.
        1.1.1. board.py: Модуль, який містить класи та функції для роботи з гральною дошкою.
        1.1.2. house.py: Модуль, який відповідає за класи та функції, пов'язані з будинками на гральній дошці.

1.1.3. indicator.py: Модуль, який містить класи та функції для індикаторів або показників у грі.

1.1.4. pause.py: Модуль для обробки паузи у грі.

1.2. tests: Тестові сценарії для перевірки функціональності гри.

1.2.1. fixtures: Фікстури, які використовуються для підготовки даних для тестів.

1.2.2. gameplay: Тестові сценарії, які перевіряють різні аспекти геймплею.

1.2.2.1. board: Тестові сценарії для перевірки роботи гральної дошки.

1.2.2.2. combo-merge: Тестові сценарії для перевірки функції комбінованого злиття.

1.2.2.3. combo_merge_reward: Тестові сценарії для перевірки винагороди після комбінованого злиття.

1.2.2.4. destructing: Тестові сценарії для перевірки руйнування об'єктів у грі.

1.2.2.5. economic: Тестові сценарії для перевірки економічних аспектів гри.

1.2.2.6. merge: Тестові сценарії для перевірки злиття об'єктів у грі.

1.2.2.7. probability: Тестові сценарії для перевірки ймовірностей подій у грі.

1.2.3. conftest.py: Модуль, який містить фікстури та налаштування для всіх тестів.

2. utils: Утилітарні функції та класи, які використовуються в проекті.

2.1. get_root.py: Модуль для отримання кореневої директорії проекту.

В классі тестування **Board** знаходиться вся логіка тестування для роботи з доскою, наприклад:

```
from utils.get_root import get_root
import yaml
from pytest_bdd import given, parsers, scenario, then, ï\when
@scenario("board.feature", "User puts the house at empty cell")
def test_board_1():
  pass
EXTRA_TYPES = {"Yml": str, "List": str}
@given(
  parsers.cfparse(
    "Game board with default preset houses {file:Yml}", extra_types=EXTRA_TYPES
  ),
  target_fixture="game_board",
)
def game_board(file,load_fixture_data):
  path = get_root(file)
  data = load_fixture_data(path)
  return data
@given(
  parsers.cfparse("User have in board queue {data:List}", extra_types=EXTRA_TYPES),
  target_fixture="spawn_queue",
)
def spawn_queue(data):
  raw_data = data[1:-1].split(",")
  data = [int(i) for i in raw_data]
  return {"spawn_queue": data}
@when(
  parsers.cfparse("User place on {data:List}", extra_types=EXTRA_TYPES),
  target_fixture="game_data",
)
def user_place(data, game_board, spawn_queue):
  raw_data = data[1:-1].split(",")
  user_place = [int(i) for i in raw_data]
  x, y = user_place
  board = game_board.get("board")
  spawn_queue = spawn_queue.get("spawn_queue")
```

```
house = spawn_queue[3]
assert house == spawn_queue[3]
board[x][y] = house
assert board[x][y] == house
return [board, user_place, house]
```

Таблиця 3.1.6 – Вірогідність спавну будинка n рівня на панелі спавн

Вірогідність спавну будинка n рівня на панелі спавну:										
Максимальний рівень будинка на ігровому полі:	1	2	3	4	5	6	7	8	9	10
0	1	-	-	-	-	-	-	-	-	-
1	1	-	-	-	-	-	-	-	-	-
2	1	-	-	-	-	-	-	-	-	-
3	0,7	0,3	-	-	-	-	-	-	-	-
4	0,5	0,3	0,2	-	-	-	-	-	-	-
5	0,4	0,3	0,2	0,1	-	-	-	-	-	-
6	0,35	0,3	0,2	0,1	0,05	-	-	-	-	-
7	0,33	0,3	0,2	0,1	0,05	0,02	-	-	-	-
8	0,32	0,3	0,2	0,1	0,05	0,02	0,01	-	-	-
9	0,315	0,3	0,2	0,1	0,05	0,02	0,01	-	-	-
10	0,314	0,3	0,2	0,1	0,05	0,02	0,01	-	-	-

У цьому коді представлено ключовий модуль цієї системи, який відповідає за взаємодію з пральною дошкою, розміщення об'єктів на ній та виконання користувацьких дій.

В класі тестування **Combo-Merge** знаходиться вся логіка тестування поєднання різних будівель, наприклад:

```
Feature: Combo-Merge
 Checking for merge of a multiple of houses
 @positive
 Scenario: Combo-merge function
  Given Game board with default preset houses
boards/give/combo_merge_board/positive/default.yml
  Given User have in board queue [1,1,1,1]
  When User place on [1,1]
  Then From house lvl 1 should be generated a new
boards/give/combo_merge_board/positive/result_board.yml with combo-merge result lvl 3 house
  Then User gets a crystal
 @negative
 Scenario: Negative test to check the impossibility of combo-merging houses of different
lvl
  Given Game board with default preset houses
boards/give/combo_merge_board/negative/default.yml
  Given User have in board queue [1,1,1,3]
  When User place on [1,1]
  Then From house lvl 3 should not be generated a new
boards/give/combo_merge_board/negative/result_board.yml with combo-merge result lvl 4
house
```

Цей код описує два тестові сценарії для перевірки функціональності "Combo-Merge" у грі. Перший сценарій (Scenario 1) перевіряє коректність роботи функції "Combo-merge", яка дозволяє злити кілька будинків в один більш високого рівня. У цьому сценарії вказується, що після розміщення будинка рівня 1 на дошці гри, він автоматично зливається з іншими будинками рівня 1 у новий будинок рівня 3. Користувач також отримує кристал за успішне злиття.

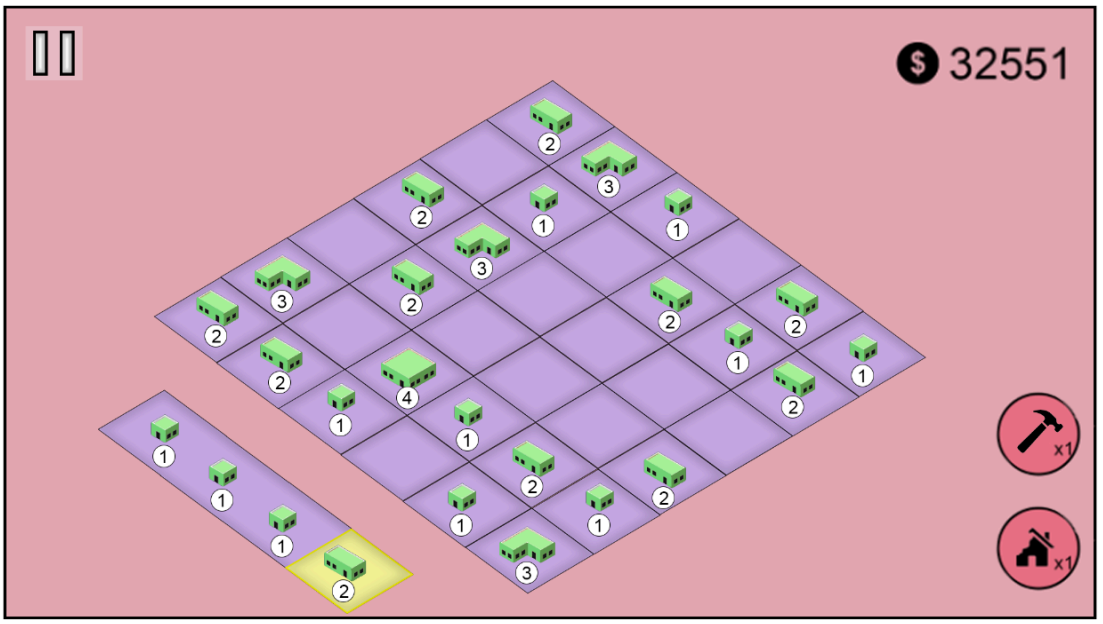

Рисунок 3.1.7 – Приклад різних механік поєднання

Другий сценарій (Scenario 2) створений для перевірки випадку, коли неможливо здійснити "Combo-merge" через наявність будинків різних рівнів. У цьому випадку, коли користувач розміщує будинок рівня 1 поруч з будинком рівня 3, злиття не відбувається, і новий будинок рівня 4 не створюється.

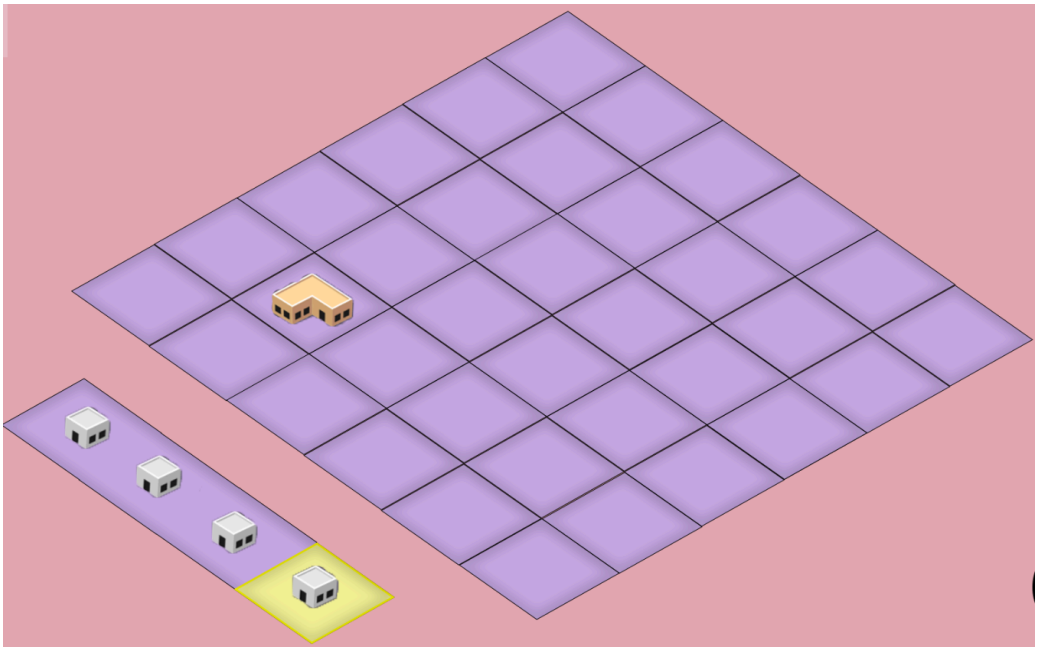

Рисунок 3.1.8 – Приклад коли поєднання будинків неможливе

В класі тестування **Economic** знаходиться вся логіка тестування економіки гри.

```
Feature: Economic
 Checking the correct payment of coins
 Background:
  Given Random numbers of houses on the game table
 @positive
 Scenario: Getting n coins for placing a house
  When User puts the house of random lvl on cell
  Then User gets n coins
 @positive
 Scenario: Getting n*20 coins for destroying a house
  When User destroys a random house
  Then User gets n*20 coins
 @positive
 Scenario: Getting n+1 coins for using crystal
  Given Crystal
  When User use crystal on a house
  Then User gets n+1 coins
 @positive
 Scenario: Getting n*x coins for merging x houses of n lvl
  When User merges x houses of n lvl
  Then User gets n*x coins
```

Цей код описує набір тестових сценаріїв для перевірки коректності роботи економічної системи гри. Ось опис кожного сценарію:

В класі тестування **Quests** знаходиться вся логіка тестування внутрігорвих квестів гри.

```
Feature: Квести
 Щоб гравці могли виконувати завдання в грі

 Scenario: Відображення індикатора квесту
  Given Гравець відкриває гру
  When Гравець розпочинає квест "Потрібно змержити 6 шт будиночків 6-го рівня за: 70 ходів"
```

Then Відображається індикатор квесту з іконкою будинка, кількістю залишених будиночків та залишеними ходами

Scenario: Показ повної інформації про квест

Given Гравець виконує квест "Потрібно змержити 6 шт будиночків 6-го рівня за: 70 ходів"

When Гравець переглядає інформацію про квест

Then На екрані відображається повна інформація про квест з усіма умовами та нагородами

Таблиця 3.1.7 – Реалізація тестування механіки квестів

Завжди один з квестів обирається рандомно рівно-вірогідно (0,0625), на **кожен 50-тий хід**, після того, як гравець отримав на полі перший Турнірний будинок.	Зробіть шляхом мержу **10 шт** будиночків 2-го рівня за: **100** ходів	+ 20 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 3-го рівня за: **100** ходів	+ 25 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 4-го рівня за: **100** ходів	+ 30 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 5-го рівня за: **100** ходів	+ 35 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 6-го рівня за: **100** ходів	+ 40 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 7-го	+ 45 шт. кристалів

	рівня за: **100** ходів	
	Зробіть шляхом мержу **10 шт** будиночків 8-го рівня за: **100** ходів	+ 50 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 9-го рівня за: **100** ходів	+ 55 шт. кристалів
	Зробіть шляхом мержу **10 шт** будиночків 10-го рівня за: **100** ходів	+ 60 шт. кристалів
	Скористайтесь знищенням будинку любого рівня **10** разів за: **50** ходів	+ 10 шт. підвищення рівня будинку
	Змержіть **25** будиночків любого рівня за: **100** ходів	+ 25 шт. кристал
	Зробіть комбо-мерж любого рівня будинку **10** разів за: **100** ходів	+ 50 шт. кристал
	Скористайтесь переміщенням любого будинку **10** разів за: **50** ходів	+ 10 шт. руйнування будинку
	Скористайтесь	+ 10 шт. переміщення

	підвищенням рівня любого рівня будинку **10** разів за: **50** ходів	будинку
	Зробіть шляхом мержу **5 шт** Турнірних будиночків любого рівня: **100** ходів	+ 50 шт. кристал
	Зробіть Турнірний будинок **N** рівня за: **100** ходів **N** = n + 5, де n - найбільший поточний рівень турнірного будинку на полі	+ 75 шт. кристал

3.2 Програмна реалізація тестувального середовища для методології

TDD

Для веб-застосунка на основі TDD можна виділити наступні потреби тестування:

1. Функціональність роутерів: Перевірка коректності обробки запитів до всіх роутерів, що відповідають за різні сервіси (pet, store, user). Важливо переконатися, що кожен роутер правильно обробляє вхідні дані та повертає очікувані результати.
2. Middleware для аутентифікації: Перевірка правильності роботи middleware, яка відповідає за аутентифікацію користувачів. Тестування має включати перевірку ідентифікації користувача та відповідь з необхідними даними в разі успішної авторизації.

3. Кешування даних: Перевірка коректності роботи механізму кешування для оптимізації швидкодії застосунка. Важливо переконатися, що дані правильно кешуються та оновлюються відповідно до змін у джерелі даних.
4. Обробка помилок: Тестування обробки помилок у всіх компонентах застосунка. Це включає перевірку коректності повернення HTTP статусів у разі виникнення помилок, а також забезпечення правильної поведінки застосунка у випадку некоректних або відсутніх даних.
5. Взаємодія з базою даних: Перевірка коректності зберігання, оновлення та вилучення даних з бази даних. Тестування має включати перевірку моделей даних, логіку доступу до бази даних та правильність виконання запитів.
6. Використання ресурсів: Перевірка коректності завантаження ресурсів (наприклад, зображень, статичних файлів) та їх відображення в інтерфейсі застосунка.
7. Тестування производительности: Перевірка продуктивності застосунка, його здатності витримувати навантаження та швидкодії реакції на запити в реальному часі.

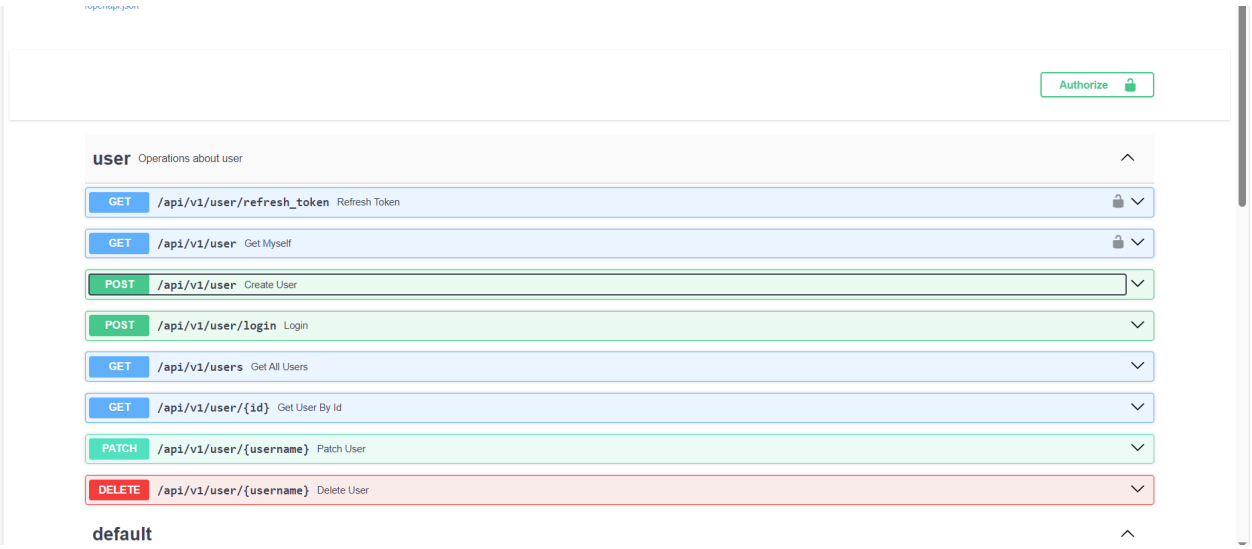

Рисунок 3.2.1 – Приклад програмного інтерфейсу

Таке тестування включає функціональне тестування для перевірки правильності роботи всіх функцій та опцій застосунка, а також тестування безпеки для виявлення та усунення можливих вразливостей, таких як SQL-ін'єкції та перехоплення сесій. Також важливим аспектом є тестування продуктивності для оцінки швидкодії застосунка та його відповідності вимогам до продуктивності.

ВИСНОВКИ

Test-Driven Development (TDD) та Behavior-Driven Development (BDD) є двома підходами до розробки програмного забезпечення, які дозволяють покращити якість коду та забезпечити відповідність функціональності вимогам користувача. В обох випадках, основна ідея полягає в написанні тестів перед фактичним кодом програми.

TDD - це методологія розробки, де розробники спочатку пишуть тести, які описують очікувану поведінку програми, а потім пишуть код, який проходить ці тести. Цей підхід сприяє створенню більш стійкого та масштабованого коду, оскільки кожна нова функція або зміна в коді супроводжується відповідними тестами. У веб-розробці TDD може бути корисним для підтримки стабільності та безпеки веб-додатків.

З іншого боку, BDD зосереджується на описі поведінки програми через зрозумілий для бізнесу мову. Сценарії поведінки описуються у вигляді "якщо-тоді" (Given-When-Then) та можуть бути легко зрозумілі та перевірені не тільки розробниками, але й бізнес-аналітиками та клієнтами. BDD допомагає забезпечити відповідність програмного продукту функціональним вимогам та очікуванням користувачів.

У веб-розробці TDD часто використовується для реалізації окремих функціональних блоків та модулів, оскільки дозволяє ефективно тестувати серверну та клієнтську логіку, забезпечуючи стабільність та надійність додатку. З іншого боку, в ігровій розробці, де акцент зазвичай робиться на складній геймплейній логіці та інтерактивності, BDD може бути кориснішим, оскільки дозволяє легко описувати та тестувати різноманітні сценарії поведінки гравців у грі.

Звісно, існують методології в програмуванні, які, подібно до Domain-Driven Design (DDD), акцентують увагу на певних аспектах розробки програмного забезпечення. Ось деякі з них:

1. Domain-Driven Design (DDD): Ця методологія розробки ставить в центр уваги область застосування програмного забезпечення (доменну модель). DDD підкреслює важливість розуміння домену бізнесу та його відображення в коді. Вона сприяє створенню гнучких, легко розширюваних та легко змінюваних систем.
2. Data-Driven Development (DataDD): Ця методологія фокусується на використанні даних як центрального елементу розробки програмного забезпечення. Вона підкреслює важливість аналізу та використання даних для прийняття рішень у процесі розробки.
3. Model-Driven Development (MDD): MDD використовує моделі як основний засіб для специфікації, проектування та реалізації програмного забезпечення. Вона спрощує розробку, дозволяючи розробникам працювати на вищому рівні абстракції, замість написання коду на низькому рівні.
4. Test-Driven Development (TDD): Хоча TDD не відноситься до методологій, спрямованих на аспекти розробки програмного забезпечення, як DDD, вона заснована на написанні тестів як першого кроку у процесі розробки. TDD сприяє покращенню якості коду та зниженню кількості помилок у програмному забезпеченні.

Ці методології можуть використовуватися окремо або в поєднанні з іншими підходами, залежно від потреб конкретного проекту та його характеристик.

ПЕРЕЛІК ПОСИЛАНЬ

1. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Thomas, D. (2001). "Manifesto for Agile Software Development". Retrieved from http://agilemanifesto.org/
2. Boehm, B., & Turner, R. (2005). "Balancing Agility and Discipline: A Guide for the Perplexed". Addison-Wesley Professional.
3. Kaner, C., Falk, J., & Nguyen, H. Q. (1999). "Testing Computer Software". John Wiley & Sons. Ambler, S. W. (2019). "Agile Modeling: Effective Practices for Extreme Programming and the Unified Process". John Wiley & Sons.
4. Cem Kaner, J. Bach, B. Pettichord (2002). "Lessons Learned in Software Testing: A Context-Driven Approach". Wiley.
5. Freeman, S., & Pryce, N. (2009). "Growing Object-Oriented Software, Guided by Tests". Addison-Wesley Professional.
6. Jeffries, R., Anderson, A., & Hendrickson, C. (2001). "Extreme Programming Installed". Addison-Wesley Professional.
7. Wynne, M., & Hellesøy, A. (2017). "The Cucumber Book: Behaviour-Driven Development for Testers and Developers". Pragmatic Bookshelf.

Додаток А – Код програми

```
import pytest
from httpx import AsyncClient
from app.services.user.logic import UserLogic
from app.services.user.models import Users
from app.services.user.schemes import UserCreate
pytestmark = pytest.mark.anyio

class TestCase:
  @pytest.mark.asyncio
  async def test_create_pet(self, client: AsyncClient, get_session):
    data = {
      "email": "string5@gmail.com",
      "first_name": "Ivan",
      "last_name": "Grechka",
      "phone": "380502906562",
      "password": "string3004",
      "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)

    res = await user_logic.create_user(password=data.password, user=data,
db=get_session)
    print(res)
    pet_data = {
      "id": 0,
      "user_id": 1,
      "name": "doggie",
      "category": {"id": 0, "name": " Dogs"},
      "status": "available",
      "tag": [
        {
          "name": "Tag"
        }
```

```
    ]
}
login_data = {
    "login": "string55",
    "password": 'string3004'
}
response = await client.post("api/v1/user/login", json=login_data)
assert response.status_code == 200

headers = {}
headers['Authorization'] = f"Bearer {response.json().get('token')}"
response = await client.post("api/v1/pet", json=pet_data, headers=headers)
assert response.status_code == 201

@pytest.mark.asyncio
async def test_delete_pet(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)

    await user_logic.create_user(password=data.password, user=data, db=get_session)

    pet_data = {
        "id": 0,
        "user_id": 1,
        "name": "doggie",
        "category": {"id": 0, "name": " Dogs"},
        "status": "available",
```

```
        "tag": [
            {
                "name": "Tag"
            }
        ]
    }
    login_data = {
        "login": "string55",
        "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200

    headers = {}
    headers['Authorization'] = f"Bearer {response.json().get('token')}"

    response = await client.post("api/v1/pet", json=pet_data, headers=headers)
    assert response.status_code == 201

    response = await client.delete("api/v1/pet/0", headers=headers)
    assert response.status_code == 200

@pytest.mark.asyncio
async def test_get_pet(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)
```

```
    await user_logic.create_user(password=data.password, user=data, db=get_session)

    pet_data = {
        "id": 0,
        "user_id": 1,
        "name": "doggie",
        "category": {"id": 0, "name": " Dogs"},
        "status": "available",
        "tag": [
            {
                "name": "Tag"
            }
        ]
    }
    login_data = {
        "login": "string55",
        "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200

    headers = {}
    headers['Authorization'] = f"Bearer {response.json().get('token')}"

    response = await client.post("api/v1/pet", json=pet_data, headers=headers)
    assert response.status_code == 201

    response = await client.get("api/v1/pet/0", headers=headers)
    assert response.status_code == 200

@pytest.mark.asyncio
async def test_get_pet_by_status(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
```

```
    "last_name": "Grechka",
    "phone": "380502906562",
    "password": "string3004",
    "login": "string55",
}
data = UserCreate(**data)
user_logic = UserLogic(Users)

await user_logic.create_user(password=data.password, user=data, db=get_session)

pet_data = {
    "id": 0,
    "user_id": 1,
    "name": "doggie",
    "category": {"id": 0, "name": " Dogs"},
    "status": "available",
    "tag": [
        {
            "name": "Tag"
        }
    ]
}

login_data = {
    "login": "string55",
    "password": 'string3004'
}
response = await client.post("api/v1/user/login", json=login_data)
assert response.status_code == 200

headers = {}
headers['Authorization'] = f"Bearer {response.json().get('token')}"

response = await client.post("api/v1/pet", json=pet_data, headers=headers)
assert response.status_code == 201
```

```
    response = await
client.get("api/v1/pet/find_by_status?status=available",headers=headers)
    assert response.status_code == 200
    assert response.json()[0]["id"] == 0
    assert response.json()[0]["user_id"] == 1
    assert response.json()[0]["name"] == "doggie"
    assert response.json()[0]["status"] == "available"

  @pytest.mark.asyncio
  async def test_get_pet_by_status_2(self, client: AsyncClient, get_session):
    data = {
      "email": "string5@gmail.com",
      "first_name": "Ivan",
      "last_name": "Grechka",
      "phone": "380502906562",
      "password": "string3004",
      "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)

    await user_logic.create_user(password=data.password, user=data, db=get_session)

    pet_data = {
      "id": 0,
      "user_id": 1,
      "name": "doggie",
      "category": {"id": 0, "name": " Dogs"},
      "status": "pending",
      "tag": [
        {
          "name": "Tag"
        }
      ]
```

```
    }

    login_data = {
        "login": "string55",
        "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200

    headers = {}
    headers['Authorization'] = f"Bearer {response.json().get('token')}"

    response = await client.post("api/v1/pet", json=pet_data, headers=headers)
    assert response.status_code == 201

    response = await client.get("api/v1/pet/find_by_status?status=pending",
headers=headers)
    assert response.status_code == 200
    assert response.json()[0]["id"] == 0
    assert response.json()[0]["user_id"] == 1


@pytest.mark.asyncio
async def test_update_pet(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)
```

```
    await user_logic.create_user(password=data.password, user=data, db=get_session)

    pet_data = {
       "id": 0,
       "user_id": 1,
       "name": "doggie",
       "tags": {"id": 0, "name": "string"},
       "category": {"id": 0, "name": " Dogs"},
       "status": "pending",
       "tag": [
          {
             "name": "Tag"
          }
       ]
    }

    login_data = {
       "login": "string55",
       "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200

    headers = {}
    headers['Authorization'] = f"Bearer {response.json().get('token')}"

    response = await client.post("api/v1/pet", json=pet_data, headers=headers)
    assert response.status_code == 201

    pet_data["name"] = "cat"
    response = await client.put("api/v1/pet/0", json=pet_data,headers=headers)
    assert response.status_code == 202
    assert response.json()["name"] == "cat"

@pytest.mark.asyncio
```

```python
async def test_update_pet_2(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }
    data = UserCreate(**data)
    user_logic = UserLogic(Users)

    await user_logic.create_user(password=data.password, user=data, db=get_session)

    pet_data = {
        "id": 0,
        "user_id": 12312,
        "name": "doggie",
        "tags": {"id": 0, "name": "string"},
        "category": {"id": 0, "name": " Dogs"},
        "status": "pending",
        "tag": [
            {
                "name": "Tag"
            }
        ]
    }
    login_data = {
        "login": "string55",
        "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200

    headers = {}
```

```
    headers['Authorization'] = f"Bearer {response.json().get('token')}"

    response = await client.post("api/v1/pet", json=pet_data, headers=headers)
    assert response.status_code == 404
    assert response.json()["detail"] == "User does not found"
```

```
import pytest
from httpx import AsyncClient
from app.services.pet.logic import PetLogic
from app.services.pet.models import Pet
from app.services.pet.schemes import PetBase
from app.services.user.logic import UserLogic
from app.services.user.models import Users
from app.services.user.schemes import UserCreate
pytestmark = pytest.mark.anyio

class TestCase:
  @pytest.mark.asyncio
  async def test_create_order(self, client: AsyncClient, get_session):
    data = {
      "email": "string5@gmail.com",
      "first_name": "Ivan",
      "last_name": "Grechka",
      "phone": "380502906562",
      "password": "string3004",
      "login": "string55",
    }
    pet_data = {
      "id": 0,
      "user_id": 1,
      "name": "doggie",
      "tags": {"id": 0, "name": "string"},
      "category": {"id": 0, "name": " Dogs"},
      "status": "available",
```

```
    "tag": [
        {
            "name": "Tag"
        }
    ]
}
user_data = UserCreate(**data)
pet_data = PetBase(**pet_data)
user_logic = UserLogic(Users)
pet_logic = PetLogic(Pet)

await user_logic.create_user(
    password=user_data.password, user=user_data, db=get_session
)
login_data = {
    "login": "string55",
    "password": 'string3004'
}
response = await client.post("api/v1/user/login", json=login_data)
assert response.status_code == 200

headers = {}
headers['Authorization'] = f"Bearer {response.json().get('token')}"

await pet_logic.create_pet(db=get_session, pet=pet_data)
order = {
    "id": 5,
    "pet_id": 0,
    "quantity": 0,
    "status": "complete",
    "complete": True,
}
```

```
    response = await client.post("api/v1/store", json=order,headers=headers)
    assert response.status_code == 201


@pytest.mark.asyncio
async def test_create_order_2(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }

    pet_data = {
        "id": 0,
        "user_id": 1,
        "name": "doggie",
        "tags": {"id": 0, "name": "string"},
        "category": {"id": 0, "name": " Dogs"},
        "status": "available",
        "tag": [
            {
                "name": "Tag"
            }
        ]
    }
    user_data = UserCreate(**data)
    pet_data = PetBase(**pet_data)
    user_logic = UserLogic(Users)
    pet_logic = PetLogic(Pet)
```

```
    await user_logic.create_user(
        password=user_data.password, user=user_data, db=get_session
    )
    await pet_logic.create_pet(db=get_session, pet=pet_data)
    order = {
        "id": 5,
        "pet_id": 1,
        "quantity": 0,
        "status": "complete",
        "complete": True,
    }
    login_data = {
        "login": "string55",
        "password": 'string3004'
    }
    response = await client.post("api/v1/user/login", json=login_data)
    assert response.status_code == 200
    headers = {}
    headers['Authorization'] = f"Bearer {response.json().get('token')}"
    response = await client.post("api/v1/store", json=order,headers=headers)
    assert response.status_code == 404
@pytest.mark.asyncio
async def test_delete_order(self, client: AsyncClient, get_session):
    data = {
        "email": "string5@gmail.com",
        "first_name": "Ivan",
        "last_name": "Grechka",
        "phone": "380502906562",
        "password": "string3004",
        "login": "string55",
    }
```