МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА - Российский технологический университет»**

# РТУ МИРЭА

Институт информационных технологий

Кафедра математического обеспечения и стандартизации информационных технологий

## ОТЧЕТ

## ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 3

**по дисциплине**

«Тестирование и верификация программного обеспечения»

Выполнил студент группы *ИКБО-74-23*

*Ермоленко В.М.*

Принял Ильичев Г.П.

Практическая работа выполнена «31» октября 2025 г. _____

«Зачтено» «__»_____ 2025 г. _____

Москва 2025

## 1. Введение.

**Цель работы:** Изучение и практическое применение подходов к разработке программного обеспечения, основанных на тестировании (TDD, ATDD, BDD, SDD), для повышения качества, надёжности и поддерживаемости кода.

Задачи работы:

1. Изучить теоретические основы методологий TDD, ATDD, BDD и SDD.
2. Реализовать практический пример для каждого метода.
3. Проанализировать влияние интеграции тестирования на архитектуру и качество программного продукта.
4. Подготовить итоговый отчёт с выводами по проделанной работе.

**Вариант задания**: 30

Название: Приложение для заметок

Функции: Создание заметки, редактирование, удаление, поиск по заметкам.

## 2. Теоретический раздел

- Test-Driven Development (TDD) — Разработка через тестирование. Это методология, при которой тесты для новой функциональности пишутся до написания самого кода. Разработка ведётся короткими циклами «Красный» (тест не проходит) → «Зелёный» (пишется минимальный код для прохождения теста) → «Рефакторинг» (код улучшается без изменения функциональности).

- Acceptance Test-Driven Development (ATDD) — Разработка через приёмочные тесты. Этот подход расширяет TDD, фокусируясь на требованиях конечного пользователя. Вся команда (заказчик, аналитики, разработчики, тестировщики) совместно определяет критерии приёмки в виде тестов, которые описывают, как система должна работать с точки зрения бизнеса.

- Behavior-Driven Development (BDD) — Разработка через поведение. BDD является развитием TDD и ATDD. Основная идея — описывать поведение системы на естественном, понятном для всех языке с использованием структуры Given-When-Then (Дано-Когда-Тогда). Эти описания служат одновременно и документацией, и основой для автоматизированных тестов.

- Specification by Example (SDD) — Спецификация на примерах. Этот подход использует конкретные, реальные примеры для формулирования требований. Вместо абстрактных правил создаются таблицы с входными данными и ожидаемыми результатами, что устраняет двусмысленность и служит живой документацией и основой для тестов.

## 3. Практическая часть

### 3.1. Этап 1: Реализация с помощью TDD (Test-Driven Development)

Разработка началась с создания теста, описывающего ожидаемое поведение функции добавления заметки. Тест test_add_note проверяет, что после вызова метода add_note созданная заметка появляется в списке.


Рис. 1 – Тест test_add_note

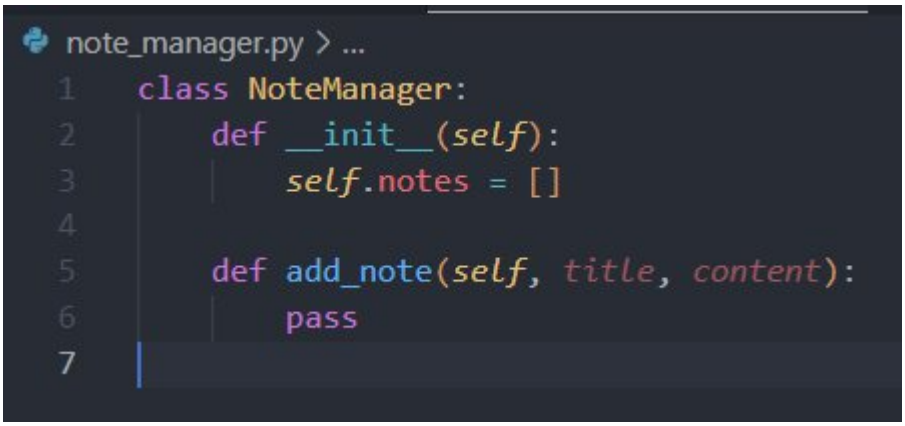


Рис. 2 – Начальное состояние класса NoteManager

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
F
======================================================================
FAIL: test_add_note (test_note_manager.TestNoteManager.test_add_note)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\User\Downloads\tiv3\test_note_manager.py", line 12, in test_add_note
    self.assertIn(note, self.manager.notes)
AssertionError: None not found in []

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 3 – Результат теста

Запуск тестов приводил к ожидаемому провалу, так как add_note ничего не добавляет в список self.notes.

### 3.1.2 Минимальная реализация для прохождения теста

Далее был написан минимальный код для того, чтобы тест test_add_note успешно прошёл.

note_manager.py > Note > __init__ > content

```python
class Note:
    def __init__(self, title, content):
        self.title = title
        self.content = content


class NoteManager:
    def __init__(self):
        self.notes = []

    def add_note(self, title, content):
        note = Note(title, content)
        self.notes.append(note)
        return note

```

Рис. 4 – Минимальная реализация add_note

После этого изменения тест был запущен повторно и успешно пройден.


Рис. 5 – Результат теста

### 3.1.3 Рефакторинг

На текущем этапе код работает только в памяти. Требуется добавить персистентность - сохранение заметок в файл. Это является шагом рефакторинга, так как мы улучшаем внутреннюю реализацию, не меняя публичный интерфейс метода add_note. Была добавлена функциональность сохранения и загрузки заметок из JSON-файла.

note_manager.py > NoteManager > add_note

```python
import uuid
import json
from datetime import datetime
import os

class Note:
    def __init__(self, title, content, note_id=None, created_at=None):
        self.id = note_id if note_id else str(uuid.uuid4())
        self.title = title
        self.content = content
        self.created_at = created_at if created_at else datetime.now().isoformat()

    def to_dict(self):
        return {
            "id": self.id, "title": self.title,
            "content": self.content, "created_at": self.created_at,
        }

    @staticmethod
    def from_dict(data):
        return Note(
            title=data["title"], content=data["content"],
            note_id=data["id"], created_at=data["created_at"],
        )

class NoteManager:
    def __init__(self, filepath="notes.json"):
        self.filepath = filepath
        self.notes = self._load_notes()

    def _load_notes(self):
        if not os.path.exists(self.filepath):
            return []
        try:
            with open(self.filepath, "r", encoding="utf-8") as f:
                data = json.load(f)
                return [Note.from_dict(note_data) for note_data in data]
        except (json.JSONDecodeError, FileNotFoundError):
            return []

    def _save_notes(self):
        with open(self.filepath, "w", encoding="utf-8") as f:
            json.dump(
                [note.to_dict() for note in self.notes], f, indent=4, ensure_ascii=False
            )

    def add_note(self, title, content):
        note = Note(title, content)
        self.notes.append(note)
        self._save_notes()
        return note

    def get_note_by_id(self, note_id):
        for note in self.notes:
            if note.id == note_id:
                return note
        return None
```

Рис. 6 – Обновлённый код менеджера заметок

После этого изменения тест был запущен повторно и успешно пройден.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 7 – Результат теста

### 3.1.4 Реализация функции удаления заметки (delete_note)

Был написан новый тест test_delete_note. Он проверяет, что после удаления заметка исчезает из менеджера.

test_note_manager.py > TestNoteManager

```python
import unittest
from note_manager import NoteManager


class TestNoteManager(unittest.TestCase):
    def setUp(self):
        self.manager = NoteManager()

    def test_add_note(self):
        note = self.manager.add_note("Покупки", "Молоко, хлеб")

        self.assertIn(note, self.manager.notes)

    def test_delete_note(self):
        note_to_delete = self.manager.add_note("Встреча", "Встреча с клиентом в 15:00")
        note_id = note_to_delete.id

        delete_successful = self.manager.delete_note(note_id)

        self.assertTrue(delete_successful)
        self.assertIsNone(self.manager.get_note_by_id(note_id))
```

Рис. 8 – Тест test_delete_note

На момент написания теста метод delete_note отсутствовал в классе NoteManager. Запуск теста приводил к ошибке AttributeError: 'NoteManager' object has no attribute 'delete_note'.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
.E
======================================================================
ERROR: test_delete_note (test_note_manager.TestNoteManager.test_delete_note)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\User\Downloads\tiv3\test_note_manager.py", line 18, in test_delete
_note
    delete_successful = self.manager.delete_note(note_id)
                        ^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'NoteManager' object has no attribute 'delete_note'

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (errors=1)
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 9 – Результат теста

### 3.1.5 Минимальная реализация

В класс NoteManager был добавлен метод delete_note, который находит заметку, удаляет её и сохраняет изменения в файл.

```
64
65      def delete_note(self, note_id):
66          note_to_delete = self.get_note_by_id(note_id)
67          if note_to_delete:
68              self.notes.remove(note_to_delete)
69              self._save_notes()
70              return True
71          return False
72
```

Рис. 10 - Реализация delete_note в NoteManager

После добавления этого метода все тесты, включая новый, были запущены и успешно пройдены.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 11 – Результат теста

### 3.1.6 Реализация функции редактирования заметки (update_note)

Для реализации функциональности редактирования был написан тест test_update_note. Он проверяет, что после вызова метода update_note содержимое заметки изменяется.

```
23      def test_update_note(self):
24          note_to_update = self.manager.add_note("Статья",
            "Черновик статьи")
25          note_id = note_to_update.id
26
27          self.manager.update_note(note_id,
            new_content="Финальная версия статьи.")
28
29          updated_note = self.manager.get_note_by_id(note_id)
30          self.assertIsNotNone(updated_note)
31          self.assertEqual(updated_note.content, "Финальная
            версия статьи.")
```

Рис. 12 – Тест test_delete_note

На момент написания теста метод update_note отсутствовал в классе NoteManager. Запуск теста приводил к ошибке AttributeError: 'NoteManager' object has no attribute 'update_note'.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
..E
======================================================================
ERROR: test_update_note (test_note_manager.TestNoteManager.test_update_note)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\User\Downloads\tiv3\test_note_manager.py", line 27, in test_update
_note
    self.manager.update_note(note_id, new_content="Финальная версия статьи.")
    ^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'NoteManager' object has no attribute 'update_note'

----------------------------------------------------------------------
Ran 3 tests in 0.004s

FAILED (errors=1)
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 13 – Результат теста

### 3.1.7 Минимальная реализация

В класс NoteManager был добавлен метод update_note, который находит заметку по ID, обновляет её поля и сохраняет изменения в файл.

```
73      def update_note(self, note_id, new_title=None,
        new_content=None):
74          note_to_update = self.get_note_by_id(note_id)
75          if note_to_update:
76              if new_title is not None:
77                  note_to_update.title = new_title
78              if new_content is not None:
79                  note_to_update.content = new_content
80              self._save_notes()
81              return note_to_update
82          return None
```

Рис. 14 - Реализация update_note в NoteManager

После добавления этого метода все тесты, включая новый, были запущены и успешно пройдены.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.004s

OK
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 15 – Результат теста

### 3.2. Этап 2: ATDD — Разработка через приёмочные тесты

После реализации основной логики фокус сместился на пользовательские сценарии. С помощью ATDD были определены и автоматизированы ключевые требования к продукту.

- **Сценарий 1: Создание и сохранение заметки.**
  Пользователь создает заметку, и она остается в списке даже после перезапуска приложения. Этот сценарий проверяет интеграцию add_note и механизма персистентности.
- **Сценарий 2: Поиск заметки.**
  Пользователь вводит ключевое слово в поиск и видит в списке только те заметки, которые содержат это слово. Это требование вводит новую функциональность — поиск.

Для проверки этих сценариев был создан отдельный файл с приёмочными тестами test_acceptance.py.

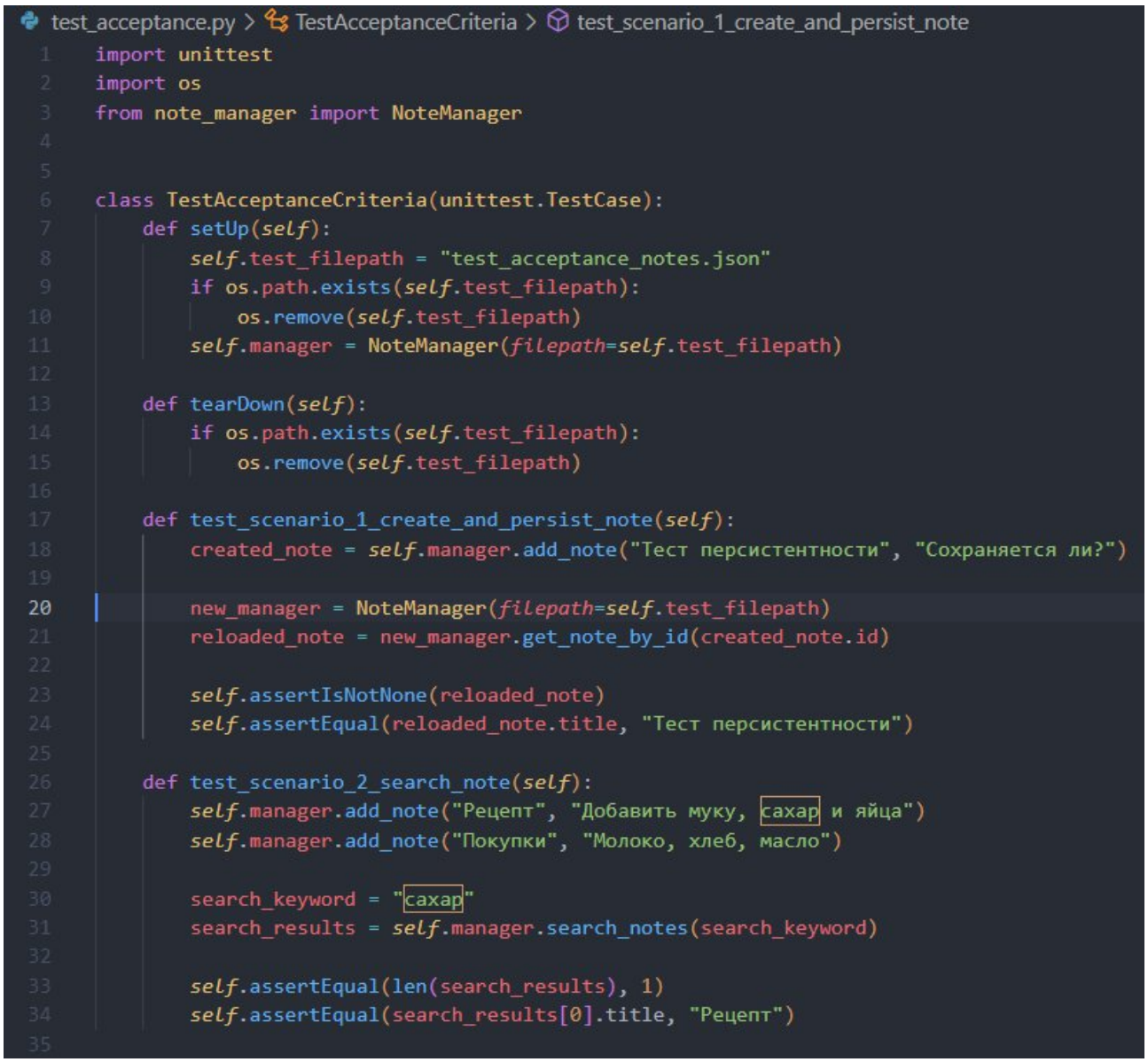


Рис. 16 - Приёмочные тесты (test_acceptance.py)

Тест test_scenario_1_create_and_persist_note прошёл сразу, так как необходимая логика уже была реализована на этапе TDD. Однако тест test_scenario_2_search_note провалился с ошибкой AttributeError, так как метод search_notes ещё не был создан.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_acceptance.py
.E
======================================================================
ERROR: test_scenario_2_search_note (test_acceptance.TestAcceptanceCriteria.test_sce
nario_2_search_note)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\User\Downloads\tiv3\test_acceptance.py", line 31, in test_scenaric
_2_search_note
    search_results = self.manager.search_notes(search_keyword)
                     ^^^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'NoteManager' object has no attribute 'search_notes'

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (errors=1)
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 17 – Результат теста

Для прохождения этого приёмочного теста в класс NoteManager был добавлен метод search_notes.

```
83
84      def search_notes(self, keyword):
85          keyword = keyword.lower()
86          return [
87              note
88              for note in self.notes
89              if keyword in note.title.lower() or keyword in note.
                content.lower()
90          ]
```

Рис. 18 – Реализация search_notes в NoteManager

После добавления метода search_notes все приёмочные тесты были запущены снова и успешно пройдены. Это подтверждает, что продукт соответствует согласованным с "заказчиком" сценариям использования.

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_acceptance.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 19 – Результат теста

### 3.3. Этап 3: BDD — Разработка через поведение

После того как ключевые пользовательские сценарии были реализованы и проверены приёмочными тестами, был применён подход BDD для создания "живой документации". Цель BDD — описать поведение системы на естественном языке, понятном как разработчикам, так и нетехническим специалистам (например, менеджерам или заказчикам).

Для этого был сформулирован сценарий поиска заметки на языке Gherkin.

features > notes_search.feature

```
Feature: Поиск заметок в приложении
  Чтобы быстро находить нужную информацию,
  Как пользователь приложения,
  Я хочу иметь возможность искать заметки по ключевому слову.

  Scenario: Успешный поиск заметки по слову в содержимом
    Given у меня есть заметка с заголовком "Рецепт" и текстом
    "Добавить муку, сахар и яйца"
    And у меня есть заметка с заголовком "Покупки" и текстом
    "Молоко, хлеб, масло"
    When я ищу по ключевому слову "сахар"
    Then я должен увидеть в результатах только одну заметку
    And заголовок этой заметки должен быть "Рецепт"
```

Рис. 20 – Сценарий BDD на языке Gherkin

Этот сценарий на естественном языке точно описывает поведение, которое было реализовано на предыдущем этапе (ATDD). Приёмочный тест test_scenario_2_search_note (Рис. 16) является прямой технической реализацией этого BDD-сценария. Его структуру можно сопоставить с Given-When-Then:

- **Given:** Строки self.manager.add_note(...) подготавливают начальное состояние системы с двумя заметками.
- **When:** Строка self.manager.search_notes("сахар") выполняет основное действие пользователя.
- **Then:** Строки self.assertEqual(...) проверяют, что результат соответствует ожиданиям, описанным в сценарии.

Таким образом, BDD-сценарий служит мостом между бизнес-требованиями и кодом, делая систему более понятной и прозрачной.

**Шаг 3.3.2: Автоматизация сценария с помощью Behave**

Для того чтобы "запустить" текстовый сценарий, используется фреймворк Behave. Он связывает каждую строчку Gherkin-сценария с функцией на Python. Для этого в папке features/steps/ был создан файл note_steps.py, содержащий "шаги" — код, реализующий логику сценария.

features > steps > note_steps.py > ...

```python
from behave import given, when, then
from note_manager import NoteManager
import os


TEST_FILEPATH = os.path.join("features", "test_notes_bdd.json")


@given('у меня есть заметка с заголовком "{title}" и текстом "{content}"')
def step_impl_add_note(context, title, content):
    if not hasattr(context, "manager"):
        if os.path.exists(TEST_FILEPATH):
            os.remove(TEST_FILEPATH)
        context.manager = NoteManager(filepath=TEST_FILEPATH)

    context.manager.add_note(title, content)

@when('я ищу по ключевому слову "{keyword}"')
def step_impl_search(context, keyword):
    context.search_results = context.manager.search_notes(keyword)

@then("я должен увидеть в результатах только одну заметку")
def step_impl_check_count(context):
    assert (
        len(context.search_results) == 1
    ), f"Ожидался 1 результат, получено {len(context.search_results)}"

@then('заголовок этой заметки должен быть "{title}"')
def step_impl_check_title(context, title):
    assert (
        context.search_results[0].title == title
    ), f"Ожидался заголовок '{title}', получен '{context.search_results[0].title}'"

```

Рис. 21 – Реализация шагов в features/steps/note_steps.py

**Шаг 3.3.3: Запуск и проверка автоматизированного сценария**

Для запуска теста достаточно выполнить команду behave в корневой папке проекта. Фреймворк автоматически найдет .feature файлы и соответствующие им шаги.

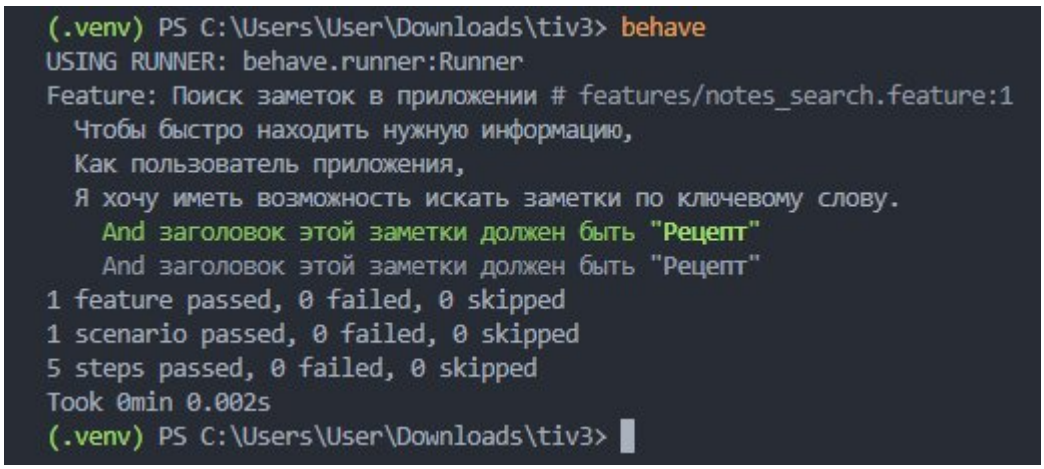


Рис. 22 – Результат успешного теста через Behave

### 3.4. Этап 4: SDD — Спецификация на примерах

Для дальнейшего уточнения работы функции поиска и для исключения любой двусмысленности был применён подход SDD. Он требует создания явных примеров, которые затем преобразуются в автоматизированные тесты. Это обеспечивает точное соответствие реализации ожиданиям. Для функции поиска была составлена следующая спецификация в виде таблицы.

*Таблица 1 — Спецификация для функции поиска на примерах*

| **Исходные заметки (Заголовок, Текст)** | **Ключевое слово для поиска** | **Ожидаемый результат (кол-во, заголовки)** |
|---|---|---|
| 1. ("Рецепт", "Добавить сахар")<br>2. ("Покупки", "Молоко") | "сахар" | (1, ["Рецепт"]) |
| 1. ("Python", "Код")<br>2. ("Задачи", "Выучить Python") | "python" | (2, ["Python", "Задачи"]) |
| 1. ("Отчет", "Поездка")<br>2. ("Идеи", "Новая поездка") | "поездк" | (2, ["Отчет", "Идеи"]) |
| 1. ("Рецепт", "Мука, сахар")<br>2. ("Спорт", "Тренировка") | "вода" | (0, []) |

На основе этой таблицы был создан автоматизированный тест. Этот тест напрямую, пример за примером, проверяет поведение системы.

```python
    def test_search_based_on_sdd_specification(self):
        self.manager.notes = []
        self.manager.add_note("Рецепт", "Добавить сахар")
        self.manager.add_note("Покупки", "Молоко")

        results_1 = self.manager.search_notes("сахар")
        self.assertEqual(len(results_1), 1)
        self.assertEqual(results_1[0].title, "Рецепт")

        self.manager.notes = []
        self.manager.add_note("Python", "Код")
        self.manager.add_note("Задачи", "Выучить Python")

        results_2 = self.manager.search_notes("python")
        self.assertEqual(len(results_2), 2)
        titles_2 = {note.title for note in results_2}
        self.assertEqual(titles_2, {"Python", "Задачи"})

        self.manager.notes = []
        self.manager.add_note("Отчет", "Поездка")
        self.manager.add_note("Идеи", "Новая поездка")

        results_3 = self.manager.search_notes("поездк")
        self.assertEqual(len(results_3), 2)
        titles_3 = {note.title for note in results_3}
        self.assertEqual(titles_3, {"Отчет", "Идеи"})

        self.manager.notes = []
        self.manager.add_note("Рецепт", "Мука, сахар")
        self.manager.add_note("Спорт", "Тренировка")

        results_4 = self.manager.search_notes("вода")
        self.assertEqual(len(results_4), 0)

```

Рис. 23 – Тест, реализующий спецификацию на примерах

```
(.venv) PS C:\Users\User\Downloads\tiv3> python -m unittest test_note_manager.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.003s

OK
(.venv) PS C:\Users\User\Downloads\tiv3>
```

Рис. 24 – Результат теста

Запуск этого теста подтверждает, что реализация функции search_notes полностью соответствует спецификации, описанной в таблице. Таким образом, SDD позволяет создать точную, проверяемую и живую документацию, которая напрямую связана с кодом через автоматизированные тесты.

## 4. Результаты тестирования и анализа

4.1. **Применение методологии TDD (Test-Driven Development)**

- В рамках TDD процесс разработки строился по циклу *«красный — зеленый — рефакторинг»*.
- Сначала формулировались юнит-тесты, которые изначально завершались неуспешно.
- Затем реализовывалась минимальная логика, обеспечивающая прохождение тестов.
- После успешного выполнения тестов код подвергался рефакторингу при сохранении корректности поведения.
- Такой подход позволил выявлять ошибки на ранних этапах и гарантировать, что каждая новая функция системы сопровождалась проверкой.

4.2. **Применение методологии ATDD (Acceptance Test-Driven Development)**

- В ATDD ключевым элементом стало согласование критериев приемки с «заказчиком» до начала реализации.
- Критерии были зафиксированы в виде приемочных тестов, которые служили ориентиром для разработчиков и тестировщиков.

4.3. **Применение методологии BDD (Behavior-Driven Development)**

- BDD позволила описывать поведение системы на естественном языке с использованием формата Gherkin.
- Сценарии вида *Given-When-Then* обеспечили прозрачность требований и сделали спецификации понятными как для разработчиков, так и для пользователей.

4.4. **Применение методологии SDD (Specification by Example / Specification-Driven Development)**

- В SDD спецификации фиксировались в виде таблиц примеров, связывающих входные данные и ожидаемые результаты.
- Эти таблицы стали основой для автоматизированных тестов и позволили устранить двусмысленность требований.

4.5. **Общий анализ результатов**

- Все методологии продемонстрировали эффективность интеграции тестирования в процесс разработки.
- TDD обеспечила надежность на уровне модулей.
- ATDD позволила согласовать ожидания с точки зрения бизнеса.
- BDD сделала спецификации доступными для всех участников проекта.
- SDD устраняет двусмысленность требований за счет конкретных примеров.

# 5. Заключение

## 5.1. **Общая оценка качества программного продукта**

- Программный продукт полностью реализует все заявленные функции: создание, редактирование, удаление и поиск заметок.
- Система успешно прошла все модульные и приёмочные тесты, которые были созданы на основе заранее определённых спецификаций и пользовательских сценариев. Это подтверждает корректность работы каждой функции и стабильность продукта в целом, включая сохранение данных между сессиями.

## 5.2. **Оценка качества документации**

- Документация по проекту создана в формате "живой документации" и представлена в виде BDD-сценария на естественном языке (Gherkin) и таблицы спецификаций на примерах (SDD).
- Такой подход обеспечивает полную прозрачность требований к функциональности (в частности, к поиску) и служит единым источником правды, облегчая коммуникацию между всеми участниками проекта.

## 5.3. **Выводы о проделанной работе**

- Применение комплекса методологий (TDD, ATDD, BDD, SDD) позволило выстроить процесс разработки, в котором тестирование является неотъемлемой частью, а не заключительным этапом. Это привело к созданию более надёжного и поддерживаемого кода.
- Разработанная архитектура и наличие полного набора тестов делают приложение готовым к дальнейшему развитию. Например, можно легко добавить новую функциональность, такую как поддержка тегов, форматирование текста или синхронизация, будучи уверенным, что существующая логика не будет нарушена.
- Полученный практический опыт подтверждает высокую эффективность разработки через тестирование для создания качественных программных продуктов.