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*

Тарасов А.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: 51

Название: Виртуальная доска для рисования
Функции — рисование, стирание, сохранение доски.

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 функций рисования линии, валидации границ и стирания линии.

- Тест test_draw_line проверяет, что после вызова метода draw_line созданная линия появляется на виртуальной доске.
- Тест test_validate_bounds проверяет, что после вызова метода draw_line с координатами которые нарушают границы доски, вызывается обработка ошибки
- Тест test_erase_line проверяет, что после вызова метода erase_line линия стирается.

test_tdd.py > TestTDD

```python
import unittest
from whiteboard import Whiteboard

class TestTDD(unittest.TestCase):
    def test_draw_line(self):
        """Тестируем рисование линии"""
        board = Whiteboard()
        board.draw_line(10, 20, 100, 200)
        self.assertEqual(board.get_lines_count(), 1)

    def test_validate_bounds(self):
        """Тестируем валидацию границ"""
        board = Whiteboard(800, 600)
        with self.assertRaises(ValueError):
            board.draw_line(-10, 20, 100, 200)

    def test_erase_line(self):
        """Тестируем стирание линии"""
        board = Whiteboard()
        board.draw_line(10, 20, 100, 200)
        board.erase_line(0)
        self.assertEqual(board.get_lines_count(), 0)

if __name__ == '__main__':
    unittest.main()
```

Рис. 1 – Тест test_tdd

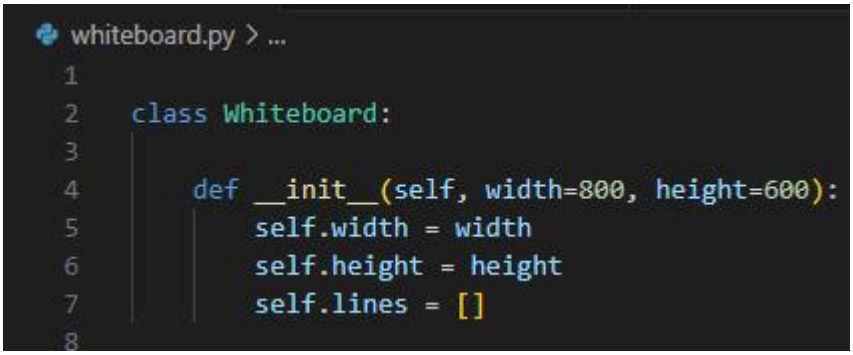


Рис. 2 – Начальное состояние класса Whiteboard

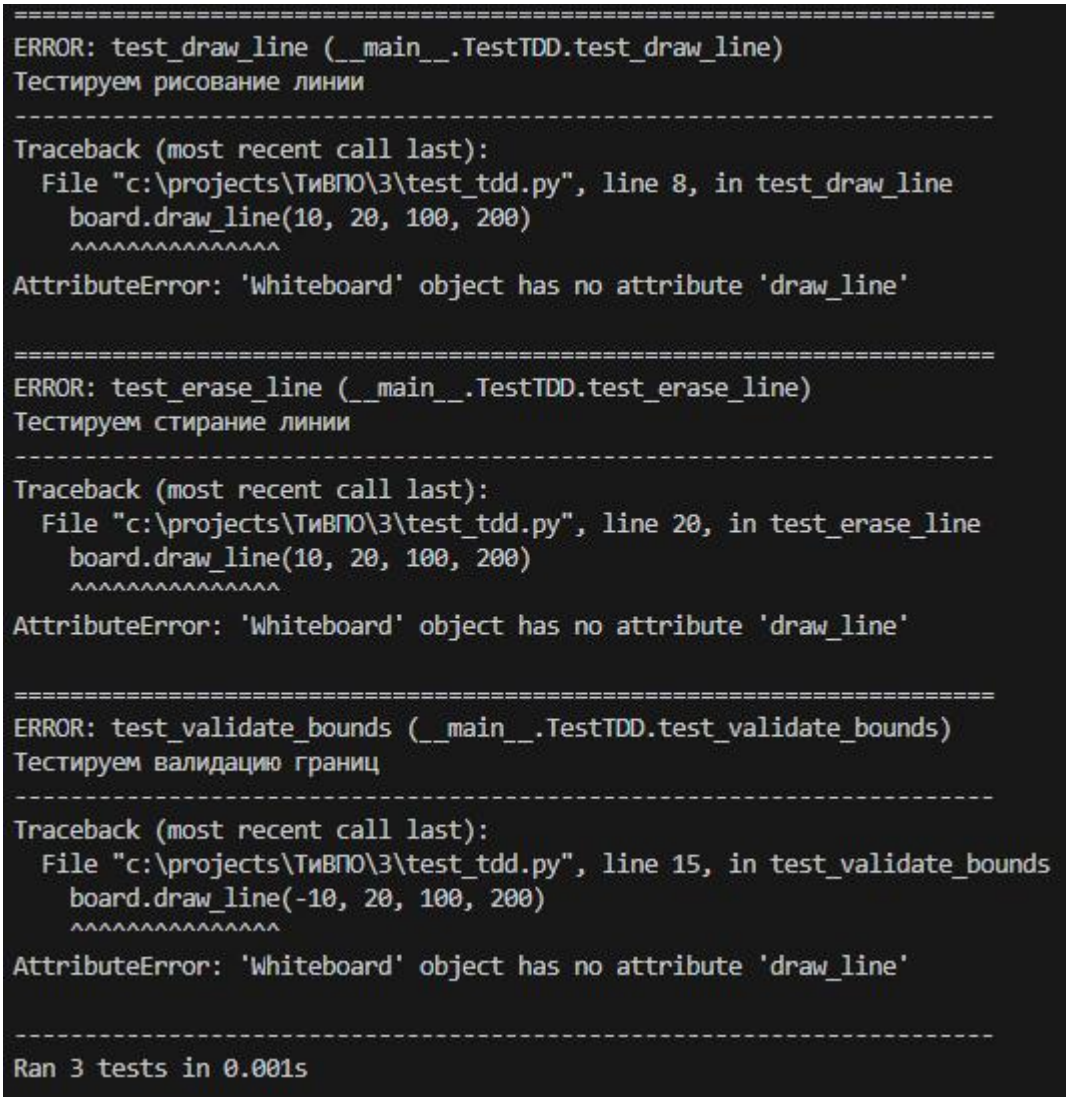


Рис. 3 – Результат теста

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_tdd успешно прошёл.

whiteboard.py > ...

```python

class Whiteboard:

    def __init__(self, width=800, height=600):
        self.width = width
        self.height = height
        self.lines = []

    def draw_line(self, x1, y1, x2, y2):
        """Рисование с валидацией границ"""
        if not (0 <= x1 <= self.width and 0 <= x2 <= self.width):
            raise ValueError('X out of bounds')
        if not (0 <= y1 <= self.height and 0 <= y2 <= self.height):
            raise ValueError('Y out of bounds')
        self.lines.append((x1, y1, x2, y2))

    def erase_line(self, line_index):
        """Стирание линии"""
        self.lines.pop(line_index)

    def get_lines_count(self):
        """Получить количество линий"""
        return len(self.lines)
```

Рис. 4 – Минимальная реализация Whiteboard

После этого изменения тест был запущен повторно и успешно пройден.

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Рис. 5 – 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 линии.**
  Пользователь открывает доску и создает несколько линий. Система сохраняет все нарисованные линии в памяти. Этот сценарий проверяет интеграцию draw_line и механизма персистентности.
- **Сценарий 2: Сохранение работы.**
  Пользователь создаёт рисунок на доске и нажимает кнопку "Сохранить". Система создаёт файл с рисунком на диске. Это требование вводит новую функциональность — сохранение в файл.

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

test_atdd.py > ...

```python
import unittest, os
from whiteboard import Whiteboard

class TestATDD(unittest.TestCase):
    def test_drawing_requirement(self):
        """Требование: Пользователь рисует несколько линий"""
        board = Whiteboard()
        board.draw_line(10, 10, 100, 100)
        board.draw_line(100, 100, 200, 50)
        self.assertEqual(board.get_lines_count(), 2)

    def test_saving_requirement(self):
        """Требование: Пользователь сохраняет работу в файл"""
        board = Whiteboard()
        board.draw_line(10, 10, 100, 100)
        board.save('test.txt')
        self.assertTrue(os.path.exists('test.txt'))
        os.remove('test.txt')

if __name__ == '__main__':
    unittest.main()

```

Рис. 6 - Приёмочные тесты (test_atdd.py)

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

```
======================================================================
ERROR: test_saving_requirement (__main__.TestATDD.test_saving_requirement)
Сценарий 2: Сохранение работы.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\projects\ТиВПО\3\test_atdd.py", line 30, in test_saving_requirement
    board.save('test.txt')
    ^^^^^^^^^^
AttributeError: 'Whiteboard' object has no attribute 'save'

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (errors=1)
```

Рис. 7 – Результат теста

Для прохождения этого приёмочного теста в класс Whiteboard был добавлен метод save.

```
25      def save(self, filename):
26          """Сохранение доски в файл"""
27          with open(filename, 'w', encoding='utf-8') as f:
28              f.write(f'Whiteboard {self.width}x{self.height}\n')
29              f.write(f'Lines: {len(self.lines)}\n')
30              for line in self.lines:
31                  f.write(f'{line[0]},{line[1]},{line[2]},{line[3]}\n')
32          return True
33
```

Рис. 8 – Реализация search_notes в NoteManager

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

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
PS C:\projects\ТиВПО\3>
```

Рис. 9 – 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.

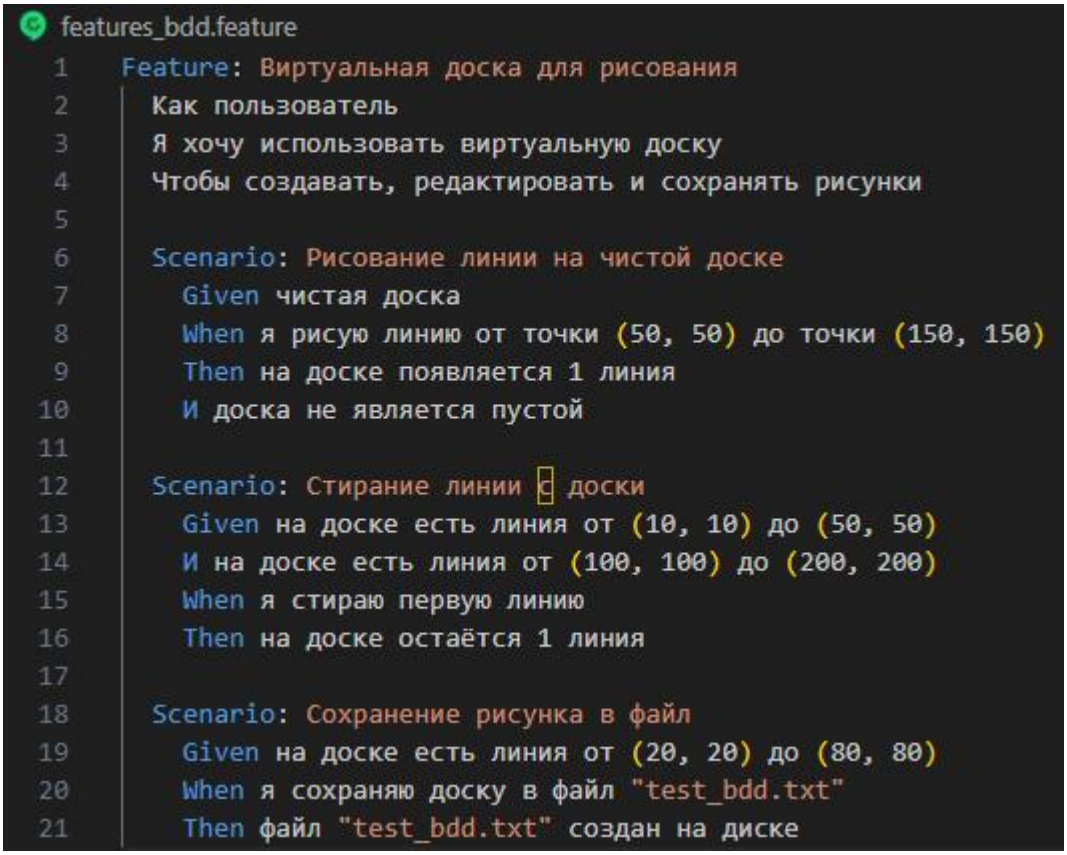


Рис. 10 – Сценарий BDD на языке Gherkin

Этот сценарий на естественном языке точно описывает поведение, которое было реализовано на предыдущем этапе (ATDD). Приёмочный тест test_saving_requirement (Рис. 6) является прямой технической реализацией этого BDD-сценария. Его структуру можно сопоставить с Given-When-Then:

- Given: Создан объект Whiteboard с размерами 800x600 и нарисованы несколько линий
- When: Вызывается метод save("whiteboard.txt")
- Then: Файл успешно создается и содержит корректные данные о размерах доски и всех линиях

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 whiteboard_steps.py, содержащий "шаги" — код, реализующий логику сценария.

features > steps > whiteboard_steps.py > ...

```python
from behave import given, when, then
from whiteboard import Whiteboard
import os


@given('чистая доска')
def step_clean_board(context):
    context.board = Whiteboard()

@when('я рисую линию от точки ({x1:d}, {y1:d}) до точки ({x2:d}, {y2:d})')
def step_draw_line(context, x1, y1, x2, y2):
    context.board.draw_line(x1, y1, x2, y2)

@then('на доске появляется {count:d} линия')
@then('на доске остаётся {count:d} линия')
def step_check_lines_count(context, count):
    assert context.board.get_lines_count() == count, \
        f"Ожидалось {count} линий, получено {context.board.get_lines_count()}"

@then('доска не является пустой')
def step_board_not_empty(context):
    assert not context.board.is_empty(), "Доска должна содержать линии"

@given('на доске есть линия от ({x1:d}, {y1:d}) до ({x2:d}, {y2:d})')
def step_add_line(context, x1, y1, x2, y2):
    if not hasattr(context, 'board'):
        context.board = Whiteboard()
    context.board.draw_line(x1, y1, x2, y2)

@when('я стираю первую линию')
def step_erase_first_line(context):
    context.board.erase_line(0)

@when('я сохраняю доску в файл "{filename}"')
def step_save_to_file(context, filename):
    context.result = context.board.save(filename)
    context.saved_file = filename

@then('файл "{filename}" создан на диске')
def step_file_exists(context, filename):
    assert os.path.exists(filename), f"Файл '{filename}' не найден"

```

Рис. 11 – Реализация шагов в features/steps/whiteboard_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.

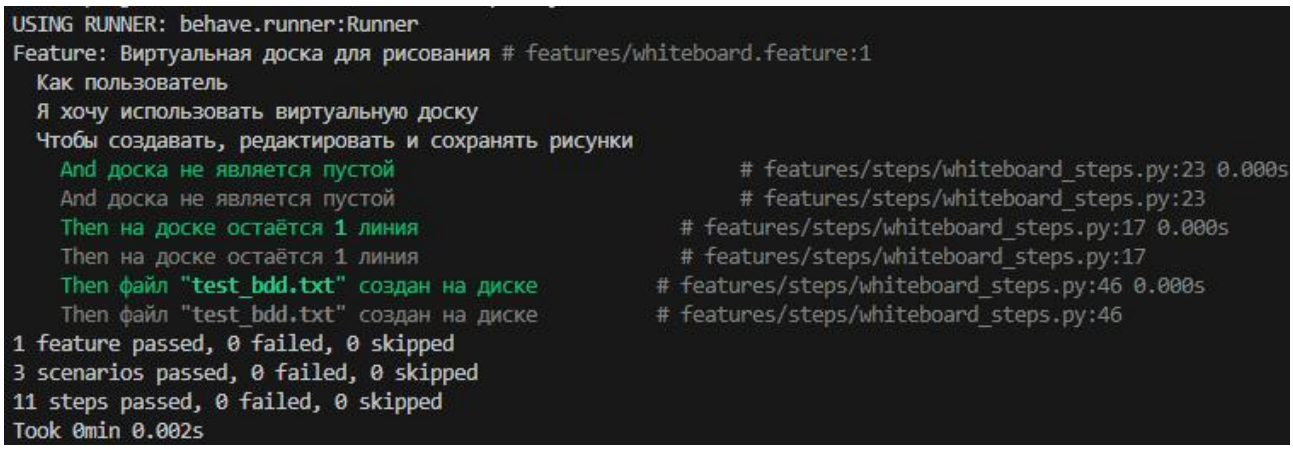


Рис. 12 – Результат успешного теста через Behave

3.4. Этап 4: SDD — Спецификация на примерах

Для дальнейшего уточнения работы функции поиска и для исключения любой двусмысленности был применён подход SDD. Он использует конкретные примеры для описания поведения системы, что особенно полезно для определения граничных случаев. Была составлена спецификация в виде таблицы.

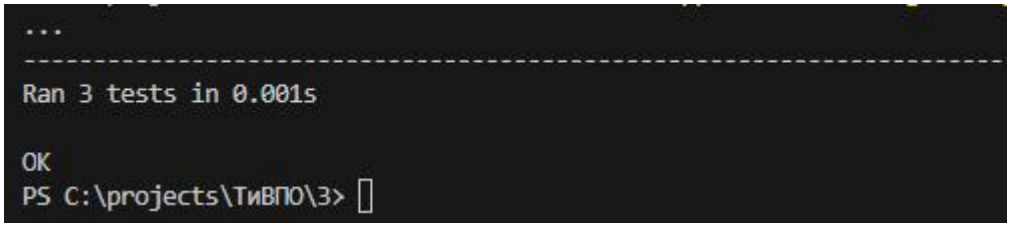
Исходное состояние доски	**Выполняемая операция**	**Ожидаемое итоговое состояние**	**Пояснение**
Пустая доска (0 линий)	draw_line(0, 0, 100, 100)	1 линия	Добавление первой линии на пустую доску
Доска с 1 линией	draw_line(50, 50, 150, 150)	2 линии	Добавление второй линии
Доска с 2 линиями	erase_line(0)	1 линия	Удаление первой линии из двух
Доска с 1 линией	save("test.txt")	Файл создан и содержит данные о доске и линии	Сохранение доски с одной линией в файл
Пустая доска (0 линий)	save("empty.txt")	Файл создан с заголовком и 0 линий	Сохранение пустой доски
Доска с 2 линиями	get_lines_count()	Возвращает 2	Проверка количества линий
Пустая доска	is_empty()	Возвращает True	Проверка пустого состояния доски
Доска с 1 линией	is_empty()	Возвращает False	Проверка непустого состояния доски

Исходное состояние доски	**Выполняемая операция**	**Ожидаемое итоговое состояние**	**Пояснение**
Любое состояние	draw_line(1000, 1000, 1200, 1200)	Вызывает ValueError ("X out of bounds" или "Y out of bounds")	Попытка рисования за границами доски (валидация)
Доска с 3 линиями	erase_line(1)	2 линии (удалена средняя линия)	Удаление линии из середины списка

Эта спецификация - основа для автоматизированных тестов. Для проверки этих примеров был создан файл whiteboard_sdd_steps.py.

```python
import unittest, os
from whiteboard import Whiteboard

class TestSDD(unittest.TestCase):
    def test_drawing_examples(self):
        board = Whiteboard()
        self.assertEqual(board.get_lines_count(), 0)
        board.draw_line(0, 0, 100, 100)
        self.assertEqual(board.get_lines_count(), 1)
        board.draw_line(50, 50, 150, 150)
        self.assertEqual(board.get_lines_count(), 2)

    def test_erasing_examples(self):
        board = Whiteboard()
        board.draw_line(10, 10, 50, 50)
        board.draw_line(60, 60, 100, 100)
        self.assertEqual(board.get_lines_count(), 2)
        board.erase_line(0)
        self.assertEqual(board.get_lines_count(), 1)

    def test_saving_examples(self):
        board = Whiteboard()
        board.draw_line(10, 10, 50, 50)
        board.save('test_sdd.txt')
        self.assertTrue(os.path.exists('test_sdd.txt'))
        os.remove('test_sdd.txt')

if __name__ == '__main__':
    unittest.main()
```

Рис. 13 – Тест, реализующий спецификацию на примерах



```
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
PS C:\projects\ТиВПО\3>
```

Рис. 14 – Результат теста

Таким образом, SDD позволяет создать точную, проверяемую и живую документацию, которая напрямую связана с кодом через автоматизированные тесты.

4. Финальное приложение полученное в ходе тестирования

whiteboard.py > ...

```python

class Whiteboard:

    def __init__(self, width=800, height=600):
        self.width = width
        self.height = height
        self.lines = []

    def draw_line(self, x1, y1, x2, y2):
        """Рисование с валидацией границ"""
        if not (0 <= x1 <= self.width and 0 <= x2 <= self.width):
            raise ValueError('X out of bounds')
        if not (0 <= y1 <= self.height and 0 <= y2 <= self.height):
            raise ValueError('Y out of bounds')
        self.lines.append((x1, y1, x2, y2))

    def erase_line(self, line_index):
        """Стирание линии"""
        self.lines.pop(line_index)

    def get_lines_count(self):
        """Получить количество линий"""
        return len(self.lines)

    def save(self, filename):
        """Сохранение доски в файл"""
        with open(filename, 'w', encoding='utf-8') as f:
            f.write(f'Whiteboard {self.width}x{self.height}\n')
            f.write(f'Lines: {len(self.lines)}\n')
            for line in self.lines:
                f.write(f'{line[0]},{line[1]},{line[2]},{line[3]}\n')
        return True

```

Р

Рис. 15 – Финальное приложение

5. Результаты тестирования и анализ

5.1. TDD (Test-Driven Development)

Применение цикла «красный → зелёный → рефакторинг» позволило обеспечить надежность базовых функций (рисование, стирание, валидация) на уровне модулей. Каждая функция сопровождалась автоматизированными тестами, что минимизировало количество ошибок.

5.2. ATDD (Acceptance Test-Driven Development)

Фокус на пользовательских сценариях (создание линий, сохранение доски) позволил согласовать реализацию с ожиданиями «заказчика». Приёмочные тесты подтвердили соответствие системы ключевым требованиям.

5.3. BDD (Behavior-Driven Development)

Использование сценариев на языке Gherkin (Given-When-Then) сделало требования прозрачными и понятными для всех участников проекта. Автоматизация через Behave обеспечила живую документацию и воспроизводимость тестов.

5.4. SDD (Specification by Example)

Конкретные примеры в табличной форме устранили неоднозначность в требованиях и стали основой для автоматизированных тестов граничных случаев (например, валидация координат, работа с пустой доской).

6. Заключение

В ходе работы были успешно применены методологии TDD, ATDD, BDD и SDD для разработки виртуальной доски. Это позволило:

- обеспечить высокую надежность кода за счет модульного и приемочного тестирования;
- улучшить коммуникацию в команде благодаря использованию единого языка спецификаций;
- создать точную и проверяемую документацию на основе примеров.

Полученный продукт реализует все заявленные функции и готов к дальнейшему расширению. Опыт подтвердил эффективность интеграции тестирования в процесс разработки для повышения качества ПО.