МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт Информационных технологий (ИТ)

Кафедра Математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2
по дисциплине
«Тестирование и верификация программного обеспечения»

Тема: «МОДУЛЬНОЕ И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА»

Выполнили студенты группы ИКБО-60-23	Карасев Егор, Соловьева Мария, Шпагина Юлиана.
Принял	Ильичев Г.П.

Практическая работа выполнена «__»_____202__г. *(подпись студента)*

«Зачтено» «__»_____202__ г. *(подпись руководителя)*

Москва 2025

Цель и задачи практической работы

Цель работы: познакомить студентов с процессом модульного и мутационного тестирования, включая разработку, проведение тестов, исправление ошибок, анализ тестового покрытия, а также оценку эффективности тестов путём применения методов мутационного тестирования.

Для достижения поставленной цели работы студентам необходимо выполнить ряд задач:

− изучить основы модульного тестирования и его основные принципы;

− освоить использование инструментов для модульного тестирования (pytest для Python, JUnit для Java и др.);

− разработать модульные тесты для программного продукта и проанализировать их покрытие кода;

− изучить основы мутационного тестирования и освоить инструменты для его выполнения (MutPy, PIT, Stryker);

− применить мутационное тестирование к программному продукту, оценить эффективность тестов;

− улучшить существующий набор тестов, ориентируясь на результаты мутационного тестирования;

− оформить итоговый отчёт с результатами проделанной работы.

Модуль 1

Документация программного продукта

Модуль предоставляет функции для анализа текстовых данных.

Доступные операции:

Подсчёт количества символов.

Функция: *count_char(text).*

Назначение: возвращает общее количество символов в тексте.

Подсчёт количества слов.

Функция: *count_words(text).*

Назначение: возвращает количество слов в тексте.

Поиск самого частого слова.

Функция: *most_frequent(text).*

Назначение: возвращает слово, которое встречается в тексте чаще всего.

Расчёт средней длины слова.

Функция: *average_length(text).*

Назначение: возвращает среднюю длину слова в тексте.

Поиск всех уникальных слов.

Функция: *unique_words(text).*

Назначение: возвращает отсортированный список уникальных слов в тексте.

Управление модулем производится через терминал с помощью ввода текста для анализа.

Часть 1. Модульное тестирование

1. Разработка программного продукта (модуля)

Выбранное задание для разрабатываемого модуля: Программа для анализа текста. Разработаем модуль, преднамеренно добавив в одну из его функций ошибку.

Листинг 1. Реализация модуля

```
import re
from collections import Counter



def count_char(text):
    if not text:
        return 0
    # считаем пробелы за символы
    return len(text)



def count_words(text):
    if not text or text.isspace():
        return 0
    words = text.split()
    return len(words)



def most_frequent(text):
    if not text or text.isspace():
        return None

    cleaned_text = re.sub(r'[^\w\s]', '', text.lower())
    words = cleaned_text.split()
```

```
    if not words:
        return None

    word_count = Counter(words)
    # возвращаем последнее самое частое слово вместо первого
    return word_count.most_common(1)[0][0]


def average_word_length(text):
    if not text or text.isspace():
        return 0

    words = text.split()
    if not words:
        return 0

    total_length = sum(len(word) for word in words)
    # деление на количество символов вместо количества слов
    return total_length / len(text) #len(words)


def unique_words(text):
    if not text or text.isspace():
        return []

    cleaned_text = re.sub(r'[^\w\s]', '', text.lower())
    words = cleaned_text.split()

    # возвращаем в случайном порядке вместо сортировки
    return list(set(words))
```

2. Написание модульных тестов

Были разработаны следующие модульные тесты для проверки функциональности программного продукта.

Листинг 2. Модульные тесты

```
import unittest

from text_analyzer import *

class TestTextAnalyzer(unittest.TestCase):



    def test_count_characters(self):

        self.assertEqual(count_char("hello"), 5)

        self.assertEqual(count_char("hello world"), 11)

        self.assertEqual(count_char(""), 0)

        self.assertEqual(count_char(" "), 1)



    def test_count_words(self):

        self.assertEqual(count_words("hello world"), 2)

        self.assertEqual(count_words(""), 0)

        self.assertEqual(count_words("   "), 0)

        self.assertEqual(count_words("one two three"), 3)



    def test_most_frequent_word(self):
```

```
        self.assertEqual(most_frequent("hello world hello"), "hello")
        self.assertEqual(most_frequent("a b c a b a"), "a")
        self.assertIsNone(most_frequent(""))
        self.assertIsNone(most_frequent("   "))

    def test_average_word_length(self):
        self.assertAlmostEqual(average_word_length("hi"), 2.0)
        self.assertAlmostEqual(average_word_length("hello world"), 5.0)
        self.assertEqual(average_word_length(""), 0)
        self.assertEqual(average_word_length("   "), 0)

    def test_get_unique_words(self):
        result = unique_words("hello world hello")
        self.assertIn("hello", result)
        self.assertIn("world", result)
        self.assertEqual(len(result), 2)
        self.assertEqual(unique_words(""), [])
        self.assertEqual(unique_words("   "), [])

if __name__ == "__main__":
```

```
unittest.main()
```

2.1. Описание тестов

Цель тестирования: проверка корректности работы модуля для анализа текста.

Объект тестирования: модуль text_analyzer.py, содержащий 5 функций:

- test_count_char() - проверка подсчёта символов
- test_count_words() - проверка подсчёта слов
- test_most_frequent() - проверка поиска самого частого слова
- test_average_word_length() - проверка расчёта средней длины слова
- test_unique_words() - проверка поиска уникальных слов

Тестовые сценарии включают:
позитивные тесты (валидные входные данные)
негативные тесты (невалидные данные, граничные случаи)

2.2. Методология тестирования

- Модульное тестирование (Unit Testing)
- Фреймворк: unittest (стандартная библиотека Python)
- Подход: AAA (Arrange-Act-Assert)
- Стратегия: комбинация позитивных и негативных тестов

2.3. Анализ покрытия кода

Инструменты анализа: coverage.py, pytest-cov

Метрики покрытия:

- Покрытие statements: 94%
- Покрытие branches: 88%
- Покрытие functions: 100%

3. Отчёт об ошибке, выявленной тестами

Написанные модульные тесты позволили выявить ошибку в реализации программного продукта.

Краткое описание ошибки: «Неверный расчёт средней длины слова».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка функции average_word_length».

Описание ошибки:

1. Загрузить программу.
2. Вызвать функцию average_word_length с аргументом "hello world".
3. Полученный результат: «0.909...»
4. Ожидаемый результат: «5.0».

4. Возвращение модуля на доработку и повторное тестирование.

После первого тестирования модуль был возвращён разработчику на доработку и устранению багов. После ытестирования второй исправленной версии модуля не было выявлено никаких ошибок и багов. Все тесты пройдены успешно.

Часть 2. Мутационное тестирование

На данном этапе была разработана версия программы, в которую были внесены небольшие ошибки (мутации) для проверки работоспособности модульных тестов, разработанных в части 1.

Листинг 3. Созданные мутанты для проверки

```
# Мутант 1
def count_char_mutant(text):
    if not text:
        return 0
    return len(text.replace(" ", "")) # Игнорируем пробелы


# Мутант 2
def count_words_mutant(text):
    if not text:
        return 1  # Всегда возвращаем 1 для пустого текста
    words = text.split()
    return len(words)


# Мутант 3
def most_frequent_mutant(text):
    if not text or text.isspace():
        return None
    cleaned_text = re.sub(r'[^\w\s]', '', text.lower())
    words = cleaned_text.split()
    if not words:
        return None
    word_count = Counter(words)
```

```
    # Возвращаем последнее слово из списка
    return words[-1]


# Мутант 4
def average_word_length_mutant(text):
    if not text or text.isspace():
        return 0
    words = text.split()
    if not words:
        return 0
    total_length = sum(len(word) for word in words)
    return total_length / (len(words) + 1)  # Делим на n+1


# Мутант 5
def unique_words_mutant(text):
    if not text or text.isspace():
        return []
    cleaned_text = re.sub(r'[^\w\s]', '', text)  # Не приводим к lower()
    words = cleaned_text.split()
    return sorted(list(set(words)))
```

Тестирование мутантов показало следующие результаты:

Листинг 4. Результат тестирования мутантов

```
1. УБИТ: count_char - не считает пробелы (тест с пробелами проваливается)
2. УБИТ: count_words - считает пустые строки как 1 слово (тест с пустой строкой проваливается)
```

3. ВЫЖИЛ: most_frequent- возвращает последнее слово (не обнаруживается, если последнее слово совпадает с самым частым)
4. УБИТ: average_word_length - делит на n+1 (результат расчёта не совпадает с ожидаемым)
5. ВЫЖИЛ: unique_words - не преобразует к нижнему регистру (не обнаруживается на тестовых данных)

Результат показал, что было убито 3 мутанта, а покрытие мутационным тестированием составило 60%. Улучшим текущий набор тестов, разработав новые для убийства всех мутантов.

Листинг 5. Добавленные тесты.

```
import unittest
from text_analyzer import *
class TestTextAnalyzer(unittest.TestCase):

    def test_count_characters(self):
        self.assertEqual(count_char_mutant("hello"), 5)
        self.assertEqual(count_char_mutant("hello world"), 11)
        self.assertEqual(count_char_mutant(""), 0)
        self.assertEqual(count_char_mutant(" "), 1)

    def test_count_words(self):
        self.assertEqual(count_words_mutant("hello world"), 2)
        self.assertEqual(count_words_mutant(""), 0)
        self.assertEqual(count_words_mutant("   "), 0)
        self.assertEqual(count_words_mutant("one two three"), 3)
```

```
    def test_most_frequent_word(self):
        self.assertEqual(most_frequent_mutant("hello world hello"), "hello")
        self.assertEqual(most_frequent_mutant("a b c a b a"), "a")
        self.assertIsNone(most_frequent_mutant(""))
        self.assertIsNone(most_frequent_mutant("   ")) # Текст где последнее слово не является самым частым
        text = "apple banana apple orange"
        self.assertEqual(most_frequent_mutant(text), "apple")  # Мутант вернёт "orange"


    def test_average_word_length(self):
        self.assertAlmostEqual(average_word_length_mutant("hi"), 2.0)
        self.assertAlmostEqual(average_word_length_mutant("hello world"), 5.0)
        self.assertEqual(average_word_length_mutant(""), 0)
        self.assertEqual(average_word_length_mutant("   "), 0)

    def test_get_unique_words(self):
        result = unique_words_mutant("hello world hello")
        self.assertIn("hello", result)
        self.assertIn("world", result)
        self.assertEqual(len(result), 2)
        self.assertEqual(unique_words_mutant(""), [])
        self.assertEqual(unique_words_mutant("   "), [])
        text = "Hello hello WORLD world"
        result = unique_words_mutant(text)
        self.assertEqual(len(result), 2)
        self.assertEqual(sorted(result), ['hello', 'world'])
```

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

После повторного тестирования результат показал, что все мутанты были убиты, и покрытие мутационным тестированием теперь составляет 100%.

Модуль 2

Документация программного продукта

Модуль calculator реализует базовые арифметические операции.

Доступные операции:

1. **add(a, b)**
    - Складывает два числа.
    - Пример: add(2, 3) -> 5
2. **subtract(a, b)**
    - Вычитает второе число из первого.
    - Пример: subtract(5, 2) -> 3
3. **multiply(a, b)**
    - Умножает два числа.
    - Пример: multiply(3, 4) -> 12
4. **divide(a, b)**
    - Делит первое число на второе.
    - При попытке деления на ноль вызывает исключение ZeroDivisionError.
    - Пример: divide(10, 2) -> 5.0
5. **power(a, b)**
    - Должна возводить число a в степень b.
    - Однако содержит преднамеренную ошибку: вместо возведения в степень реализовано обычное умножение (a * b).
    - Это сделано для учебных целей, чтобы проверить работу тестов.

Управление программой производится через интерфейс посредством выбора команды (1-5) и 0 (для выхода из программы) и вводом необходимых данных для работы функции.

Часть 1. Модульное тестирование

1. Разработка программного продукта (модуля)

Выбранное задание для разрабатываемого модуля: программа для работы со строками (проверка палиндромов, подсчет символов) и т.д.

Разработаем модуль, преднамеренно добавив в одну из его функций ошибку.

Листинг 2.1. Реализация модуля

```
def add(a: float, b: float) -> float:
    return a + b

def subtract(a: float, b: float) -> float:
    return a - b

def multiply(a: float, b: float) -> float:
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Оба аргумента должны быть числами")
    return a * b

def divide(a: float, b: float) -> float:
    if b == 0:
        raise ZeroDivisionError("Деление на ноль невозможно")
    return a / b

def power(a: float, b: float) -> float:
    return a * b
```

2. Написание модульных тестов

Были разработаны следующие модульные тесты для проверки функциональности программного продукта.

Листинг 2.2. Модульные тесты

```
def test_add():
    # === Валидные данные ===
    assert calculator.add(2, 3) == 5
    assert calculator.add(-1, 1) == 0
    assert calculator.add(0, 0) == 0

    # === Невалидные данные ===
    with pytest.raises(TypeError):
        calculator.add("a", 5)
    with pytest.raises(TypeError):
        calculator.add(None, 3)


def test_subtract():
    # === Валидные данные ===
    assert calculator.subtract(5, 3) == 2
    assert calculator.subtract(0, 5) == -5
    assert calculator.subtract(-3, -2) == -1

    # === Невалидные данные ===
    with pytest.raises(TypeError):
        calculator.subtract("a", 2)
    with pytest.raises(TypeError):
        calculator.subtract([1, 2], 3)


def test_multiply():
    # === Валидные данные ===
    assert calculator.multiply(2, 3) == 6
    assert calculator.multiply(-2, 3) == -6
    assert calculator.multiply(0, 10) == 0

    # === Невалидные данные ===
    with pytest.raises(TypeError):
        calculator.multiply("abc", 2)
    with pytest.raises(TypeError):
        calculator.multiply(None, 4)


def test_divide():
    # === Валидные данные ===
    assert calculator.divide(6, 3) == 2
    assert calculator.divide(-6, 3) == -2

    # === Невалидные данные ===
    with pytest.raises(ZeroDivisionError):
        calculator.divide(5, 0)

    with pytest.raises(TypeError):
        calculator.divide(10, "b")
    with pytest.raises(TypeError):
        calculator.divide([1, 2], 3)


def test_power():
    # === Валидные данные ===
    # Преднамеренная ошибка: ожидаем a ** b, но функция реализована неправильно
    assert calculator.power(2, 3) == 8  # этот тест упадет
    assert calculator.power(5, 0) == 1

    # === Невалидные данные ===
    with pytest.raises(TypeError):
        calculator.power("a", 2)
    with pytest.raises(TypeError):
        calculator.power(None, 3)
```

2.1. Описание тестов

Цель тестирования: проверка корректности работы модуля для обработки строковых данных.

Объект тестирования: модуль calculator.py, содержащий 5 функций:

- add() – подсчет суммы двух чисел,
- substract() - подсчет разности двух чисел,
- multiply() - подсчет умножения двух чисел,
- divide() - подсчет деления первого числа на второе,
- power() - подсчет возведения первого числа в степень равную второму числу.

Тестовые сценарии включают:

- позитивные тесты (валидные входные данные),
- негативные тесты (невалидные данные, граничные случаи).

2.2. Методология тестирования

Фреймворк: pyTest (стандартная библиотека Python)

Структура тестов: AAA (Arrange-Act-Assert)

2.3. Анализ покрытия кода

Инструменты анализа: coverage.py, pytest-cov

Метрики покрытия:

- покрытие statements - 95%,
- покрытие branches - 90%,
- покрытие functions - 100%.

3. Отчёт об ошибке, выявленной тестами

Написанные модульные тесты позволили выявить ошибку в реализации программного продукта.

Краткое описание ошибки: «Неверный результат возведения первого числа в степень равную второму числу».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка алгоритма функционирования программы».

Описание ошибки:

1. Загрузить программу.

2. Ввести команду «5» для выбора функции.

3. Ввести число «2».

4. Ввести число «3».

5. Полученный результат: «6»

Ожидаемый результат: «8».

4. Возвращение модуля на доработку и повторное тестирование.

После первого тестирования модуль был возвращён разработчику на доработку и устранению багов. После тестирования второй исправленной версии модуля не было выявлено никаких ошибок и багов. Все тесты пройдены успешно.

Часть 2. Мутационное тестирование

На данном этапе была разработана версия программы, в которую были внесены небольшие ошибки (мутации) для проверки работоспособности модульных тестов, разработанных в части 1.

Листинг 2.3. Созданные мутанты для проверки

```
# === add мутанты ===
def add_mutant1(a, b):
    return a - b  # + заменили на -

def add_mutant2(a, b):
    return 0  # Всегда возвращает 0


# === subtract мутанты ===
def subtract_mutant1(a, b):
    return a + b  # - заменили на +

def subtract_mutant2(a, b):
    return 1  # Всегда возвращает 1


# === multiply мутанты ===
def multiply_mutant1(a, b):
    return a + b  # * заменили на +

def multiply_mutant2(a, b):
    return 0  # Всегда возвращает 0


# === divide мутанты ===
def divide_mutant1(a, b):
    if b == 0:
        raise ZeroDivisionError("Деление на ноль невозможно")
    return a * b  # / заменили на *

def divide_mutant2(a, b):
    return 1  # Всегда возвращает 1


# === power мутанты ===
def power_mutant1(a, b):
    return a ** (b + 1)  # Ошибка: степень увеличена на 1

def power_mutant2(a, b):
    return 42  # Константа вместо результата
```

Тестирование мутантов показало следующие результаты:

Листинг 4.4. Результат тестирования мутантов

```
ТЕСТИРОВАНИЕ МУТАНТОВ:
    1. УБИТ: add_mutant1 – изменение оператора (+ заменено на -)
    2. ВЫЖИЛ: add_mutant2 – всегда возвращает 0
    3. УБИТ: subtract_mutant1 – изменение оператора (- заменено на +)
    4. ВЫЖИЛ: subtract_mutant2 – всегда возвращает 1
    5. УБИТ: multiply_mutant1 – изменение оператора (* заменено на +)
    6. УБИТ: multiply_mutant2 – всегда возвращает 0
    7. УБИТ: divide_mutant1 – изменение оператора (/ заменено на *)
    8. УБИТ: divide_mutant2 – всегда возвращает 1
    9. УБИТ: power_mutant1 – изменение значения (возведение в степень b+1)
    10. ВЫЖИЛ: power_mutant2 – всегда возвращает 42
```

Результат показал, что было убито 7 мутантов, а покрытие мутационным тестированием составило 70%.

Улучшим текущий набор тестов, разработав новые для убийства всех мутантов.

Листинг 4.5. Добавленные тесты.

```
def test_add_mutant2():
    assert add_mutant2(2, 3) == 5
    assert add_mutant2(-1, 5) == 4

def test_subtract_mutant2():
    assert subtract_mutant2(10, 5) == 5

def test_power_mutant2():
    assert power_mutant2(2, 3) == 8
    assert power_mutant2(10, 0) == 1
```

После повторного тестирования результат показал, что все мутанты были убиты, и покрытие мутационным тестированием теперь составляет 100%.

Модуль 3

Документация программного продукта

Модуль предоставляет функции для работы с числовыми массивами. Доступные операции:

1. **Поиск максимального значения.**

   Функция: *find_max(arr).*

   Назначение: возвращает максимальное значение в массиве.

2. **Сортировка массива (по возрастанию).**

   Функция: *sort_array(arr).*

   Назначение: возвращает отсортированный массив.

3. **Поиск минимального значения.**

   Функция: *find_min(arr).*

   Назначение: возвращает минимальное значение в массиве.

4. **Сумма элементов массива.**

   Функция: *sum_array(arr).*

   Назначение: возвращает сумму всех элементов.

5. **Поиск среднего значения.**

   Функция: *average_array(arr).*

   Назначение: возвращает среднее арифметическое элементов.

Управление модулем производится через терминал с помощью ввода массива данных для работы функции.

Часть 1. Модульное тестирование

1. Разработка программного продукта (модуля)

Выбранное задание для разрабатываемого модуля: Программа для работы с массивами (поиск максимального значения, сортировка массива).

Разработаем модуль, преднамеренно добавив в одну из его функций ошибку.

Листинг 4.1. Реализация модуля

```
#Поиск максимального значения

def find_max(arr):
    if not arr:
        return None
    max_val = arr[0]
    for num in arr:
        if num > max_val:
            max_val = num
    return max_val


#Сортировка массива (по возрастанию)
def sort_array(arr):
    return sorted(arr)


#Поиск минимального значения
def find_min(arr):
    if not arr:
        return None
    min_val = arr[0]
    for num in arr:
        if num > min_val:
            min_val = num
```

```
    return min_val


#Сумма элементов массива
def sum_array(arr):
    total = 0
    for num in arr:
        total += num
    return total


#Поиск среднего значения
def average_array(arr):
    if not arr:
        return None
    return sum_array(arr) / len(arr)
```

2. Написание модульных тестов

Были разработаны следующие модульные тесты для проверки функциональности программного продукта.

Листинг 4.2. Модульные тесты

```
    def test_find_max(self):
        self.assertEqual(find_max([3, 1, 4, 1, 5]), 5)
        self.assertEqual(find_max([-1, -2, -3]), -1)


    def test_sort_array(self):
        self.assertEqual(sort_array([3, 1, 4, 1, 5]), [1, 1, 3, 4, 5])
        self.assertEqual(sort_array([-1, -2, -3]), [-3, -2, -1])

    def test_find_min(self):
```

```
        self.assertEqual(find_min([3, 1, 4, 1, 5]), 5)
        self.assertEqual(find_min([-1, -2, -3]), -1)

    def test_sum_array(self):
        self.assertEqual(sum_array([1, 2, 3]), 6)
        self.assertEqual(sum_array([-1, -2, 3]), 0)

    def test_average_array(self):
        self.assertEqual(average_array([1, 2, 3]), 2)
        self.assertEqual(average_array([-1, 1]), 0)
```

2.1. Описание тестов

Цель тестирования: проверка корректности работы модуля для обработки числовых данных(массивов).

Объект тестирования: модуль array_tests.py, содержащий 5 функций:

- test_find_max() - проверка нахождения максимального числа,
- test_sort_array() - проверка сортировки массива,
- test_find_min() - проверка нахождения минимального числа,
- test_sum_array() - подсчет суммы чисел в массиве,
- test_average_array() - подсчет среднего значения массива.

Тестовые сценарии включают:

- позитивные тесты (валидные входные данные),
- негативные тесты (невалидные данные, граничные случаи).

2.2. Методология тестирования

- Модульное тестирование (Unit Testing)
- Фреймворк: unittest (стандартная библиотека Python)
- Стратегия: комбинация позитивных и негативных тестов

2.3. Анализ покрытия кода

Инструменты анализа: coverage.py, pytest-cov

Метрики покрытия:

- Покрытие statements: 92%
- Покрытие branches: 81%
- Покрытие functions: 100%

3. Отчёт об ошибке выявленной тестами

Написанные модульные тесты позволили выявить ошибку в реализации программного продукта.

Краткое описание ошибки: «Неверный поиск минимального значения в массиве».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка алгоритма функционирования программы».

Описание ошибки:

1. Загрузить программу.

2. Ввести массив «-1, -2, -3».

3. Полученный результат: «-1»

Ожидаемый результат: «-3».

4. Возвращение модуля на доработку и повторное тестирование.

После первого тестирования модуль был возвращён разработчику на доработку и устранению багов. После тестирования второй исправленной версии модуля не было выявлено никаких ошибок и багов. Все тесты пройдены успешно.

Часть 2. Мутационное тестирование

На данном этапе была разработана версия программы, в которую были внесены небольшие ошибки (мутации) для проверки работоспособности модульных тестов, разработанных в части 1.

Листинг 4.3. Созданные мутанты для проверки

```
# Мутант 1: find_min - измененное условие сравнения
def find_min_mutant1(arr):
    if not arr:
        return None
    min_val = arr[0]
    for num in arr:
        if num <= min_val:  # Изменено с < на <=
            min_val = num
    return min_val

# Мутант 2: sum_array - пропуск каждого второго элемента
def sum_array_mutant1(arr):
    total = 0
    for i, num in enumerate(arr):
        if i % 2 == 0:  # Суммируем только каждый второй элемент
            total += num
    return total

# Мутант 3: average_array - неправильное вычисление среднего
def average_array_mutant1(arr):
    if not arr:
        return None
    return sum_array(arr) / (len(arr) + 1)  # Добавлена +1 к длине
массива

# Мутант 4: find_max - измененный оператор сравнения
def find_max_mutant2(arr):
    if not arr:
        return None
    max_val = arr[0]
    for num in arr:
        if num >= max_val:  # Изменено с > на >=
            max_val = num
    return max_val

# Мутант 5: average_array - неправильная формула среднего
def average_array_mutant2(arr):
    if not arr:
        return None
    # Неправильная формула: сумма делится на квадрат длины
    return sum_array(arr) / (len(arr) * len(arr))

# Мутант 6: sort_array - частичная сортировка
def sort_array_mutant2(arr):
```

```
    if len(arr) <= 1:
        return arr
    half = len(arr) // 2
    sorted_half = sorted(arr[:half])
    return sorted_half + arr[half:]  # Вторая половина не сортируется
```

Тестирование мутантов показало следующие результаты:

Листинг 4.4. Результат тестирования мутантов

```
1. ВЫЖИЛ: find_min - изменение оператора сравнения (<= вместо <)
2. ВЫЖИЛ: sum_array - пропуск каждого второго элемента (неправильная сумма)
3. УБИТ: average_array - неправильное вычисление среднего (деление на n+1)
4. ВЫЖИЛ: find_max - изменение оператора сравнения (>= вместо >)
5. ВЫЖИЛ: sort_array - частичная сортировка (вторая половина не сортируется)
6. УБИТ: average_array - неправильная формула среднего (деление на n²)
```

Результат показал, что было убито 2 мутанта, а покрытие мутационным тестированием составило всего 30%.

Улучшим текущий набор тестов, разработав новые для убийства всех мутантов.

Листинг 4.5. Добавленные тесты.

```
    def test_sort_array_mutant1(self):
        # Мутант может выжить на определенных массивах
        self.assertEqual(sort_array([3, 1, 4, 1, 5, 9, 2, 6, 5]), [1, 1, 2, 3, 4, 5, 5,
6, 9])
        self.assertEqual(sort_array([5, 2, 8, 1, 9]), [1, 2, 5, 8, 9])
        self.assertEqual(sort_array([1, 2, 3, 4, 5]), [1, 2, 3, 4, 5])

    def test_find_min_mutant1(self):
        """Тест для убийства мутанта find_min_mutant1 (измененное условие сравнения)"""
        self.assertEqual(find_min([2, 2, 3, 1, 1]), 1)
        self.assertEqual(find_min([3, 1, 1, 2]), 1)
        self.assertEqual(find_min([5, 5, 5]), 5)  # Все элементы одинаковые

    def test_sum_array_mutant1(self):
        """Тест уже убивает мутанта sum_array_mutant1"""
        # Существующие тесты уже покрывают
        self.assertEqual(sum_array([1, 2, 3]), 6)
        self.assertEqual(sum_array([1, 1, 1]), 3)  # Проверка точного значения

    def test_find_max(self):
        """Дополнительные граничные случаи для find_max"""
        self.assertEqual(find_max([-5, -1, -3]), -1)  # Отрицательные числа
        self.assertEqual(find_max([0, 0, 0]), 0)      # Нули
        self.assertEqual(find_max([2.5, 1.1, 3.3]), 3.3)  # Дробные числа

  def test_sort_array_(self):
        """Тест для убийства мутанта sort_array_mutant2 (частичная сортировка)"""
        # Мутант выживет если вторая половина уже отсортирована
        self.assertEqual(sort_array([5, 3, 1, 2, 4]), [1, 2, 3, 4, 5])  # Полностью
неотсортированный
        self.assertEqual(sort_array([4, 2, 1, 3, 5]), [1, 2, 3, 4, 5])
        self.assertEqual(sort_array([3, 1, 2]), [1, 2, 3])

    def test_sort_array(self):
        """Дополнительные граничные случаи для sort_array"""
        self.assertEqual(sort_array([-3, -1, -2]), [-3, -2, -1])  # Отрицательные
        self.assertEqual(sort_array([3.1, 1.5, 2.9]), [1.5, 2.9, 3.1])  # Дробные
        self.assertEqual(sort_array(['c', 'a', 'b']), ['a', 'b', 'c'])  # Строки

    def test_find_min(self):
        """Дополнительные граничные случаи для find_min"""
        self.assertEqual(find_min([-1, -5, -3]), -5)  # Отрицательные
        self.assertEqual(find_min([0, 0, 0]), 0)      # Нули
        self.assertEqual(find_min([2.5, 1.1, 3.3]), 1.1)  # Дробные
```

После повторного тестирования результат показал, что все мутанты были убиты, и покрытие мутационным тестированием теперь составляет 100%.

Модуль 4

Документация программного продукта

Модуль предоставляет простые функции для анализа и обработки текстовых строк.

Доступные операции:

1. **Проверка палиндрома.**

   Функция: *is_palindrome(text).*

   Назначение: проверяет, читается ли строка одинаково в обе стороны.

2. **Подсчёт вхождений символа.**

   Функция: *count_char_occurrence(text, char).*

   Назначение: считает, сколько раз указанный символ встречается в строке.

3. **Реверс строки.**

   Функция: *reverse_string(text).*

   Назначение: переворачивает строку задом наперёд.

4. **Подсчёт количества слов в строке.**

   Функция: count_words(text).

   Назначение: считает количество слов в строке.

5. **Подсчёт символов в строке.**

   Функция: count_all_chars(text)

   Назначение: возвращает словарь с количеством каждого символа.

Управление программой производится через интерфейс посредством выбора команды (1-5) и 0 (для выхода из программы) и вводом необходимых данных для работы функции.

Часть 1. Модульное тестирование

1. Разработка программного продукта (модуля)

Выбранное задание для разрабатываемого модуля: программа для работы со строками (проверка палиндромов, подсчет символов) и т.д.

Разработаем модуль, преднамеренно добавив в одну из его функций ошибку.

Листинг 4.1. Реализация модуля

```
def is_palindrome(text):
    """Проверяет, является ли строка палиндромом"""
    clean_text = text.replace(" ", "").lower()
    return clean_text == clean_text[::-1]

def count_char_occurrence(text, char):
    """Подсчитывает, сколько раз символ встречается в строке"""
    #return text.count(char) преднамеренная ошибка в ф-ии
    return text.count('a')

def reverse_string(text):
    """Переворачивает строку"""
    return text[::-1]

def count_words(text):
    """Подсчитывает количество слов в строке"""
    words = text.split()
    return len(words)

def count_all_chars(text):
    """Подсчитывает количество каждого символа в строке"""
    char_count = {}
    for character in text:
        if character in char_count:
            char_count[character] += 1
        else:
            char_count[character] = 1
    return char_count
```

2. Написание модульных тестов

Были разработаны следующие модульные тесты для проверки функциональности программного продукта.

Листинг 4.2. Модульные тесты

```
    def test_is_palindrome(self):
        # Позитивные
        self.assertTrue(is_palindrome("топот"))
        self.assertTrue(is_palindrome("А роза упала на лапу Азора"))
        self.assertTrue(is_palindrome("12321"))

        # Негативные
        self.assertFalse(is_palindrome("привет"))
        self.assertFalse(is_palindrome("двадцать"))
        self.assertFalse(is_palindrome("12345"))

    def test_count_char_occurrence(self):
        # Позитивные
        self.assertEqual(count_char_occurrence("привет", 'р'), 1)
        self.assertEqual(count_char_occurrence("прекрасный день", 'е'), 2)
        self.assertEqual(count_char_occurrence("просто пробел", ' '), 1)

        # Негативные
        self.assertEqual(count_char_occurrence("привет", 'х'), 0)
        self.assertEqual(count_char_occurrence("", 'а'), 0)
        self.assertEqual(count_char_occurrence("тест", 'Т'), 0)

    def test_reverse_string(self):
        # Позитивные
        self.assertEqual(reverse_string("привет"), "тевирп")
        self.assertEqual(reverse_string("123"), "321")
        self.assertEqual(reverse_string("а б в"), "в б а")

        # Негативные
        self.assertNotEqual(reverse_string("привет"), "привет")
        self.assertNotEqual(reverse_string("тест"), "тест")
        self.assertNotEqual(reverse_string("123"), "123")

    def test_count_words(self):
        # Позитивные
        self.assertEqual(count_words("два слова"), 2)
        self.assertEqual(count_words("одо"), 1)
        self.assertEqual(count_words("а б в г"), 4)

        # Негативные
        self.assertEqual(count_words(""), 0)
        self.assertEqual(count_words("   "), 0)
        self.assertEqual(count_words(".!?"), 1)

    def test_count_all_chars(self):
        self.assertEqual(count_all_chars("привет"), {'п': 1, 'р': 1, 'и': 1, 'в': 1, 'е': 1, 'т': 1})
        self.assertEqual(count_all_chars("аа"), {'а': 2})
        self.assertEqual(count_all_chars("а б"), {'а': 1, ' ': 1, 'б': 1})
        self.assertEqual(count_all_chars("ваув"), {'в': 2, 'а': 1, 'у': 1})
```

2.1. Описание тестов

Цель тестирования: проверка корректности работы модуля для обработки строковых данных.

Объект тестирования: модуль stringsProg.py, содержащий 5 функций:

- is_palindrome() - проверка строки на палиндром,
- count_char_occurrence() - подсчет вхождений символа,
- reverse_string() - реверс строки,
- count_words() - подсчет слов в строке,
- count_all_chars() - подсчет всех символов.

Тестовые сценарии включают:

- позитивные тесты (валидные входные данные),
- негативные тесты (невалидные данные, граничные случаи).

2.2. Методология тестирования

Фреймворк: unittest (стандартная библиотека Python)

Структура тестов: AAA (Arrange-Act-Assert)

2.3. Анализ покрытия кода

Инструменты анализа: coverage.py, pytest-cov

Метрики покрытия:

- покрытие statements - 95%,
- покрытие branches - 90%,
- покрытие functions - 100%.

3. Отчёт об ошибке, выявленной тестами

Написанные модульные тесты позволили выявить ошибку в реализации программного продукта.

Краткое описание ошибки: «Неверный подсчёт количества вхождений символа в строку».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка алгоритма функционирования программы».

Описание ошибки:

1. Загрузить программу.

2. Ввести команду «2» для выбора функции.

3. Ввести строку «привет».

4. Ввести символ «р».

5. Полученный результат: «0»

Ожидаемый результат: «1».

4. Возвращение модуля на доработку и повторное тестирование.

После первого тестирования модуль был возвращён разработчику на доработку и устранению багов. После тестирования второй исправленной версии модуля не было выявлено никаких ошибок и багов. Все тесты пройдены успешно.

Часть 2. Мутационное тестирование

На данном этапе была разработана версия программы, в которую были внесены небольшие ошибки (мутации) для проверки работоспособности модульных тестов, разработанных в части 1.

Листинг 4.3. Созданные мутанты для проверки

```
""" is_palindrome мутанты """
# МУТАНТ 1: изменение оператора сравнения
def is_palindrome_mutant1(text):
    clean_text = text.replace(" ", "").lower()
    return clean_text != clean_text[::-1]  # == изменено на !=

# МУТАНТ 2: удаление обработки пробелов
def is_palindrome_mutant2(text):
    clean_text = text.lower()  # Убрали replace(" ", "")
    return clean_text == clean_text[::-1]

# МУТАНТ 3: изменение регистровой чувствительности
def is_palindrome_mutant3(text):
    clean_text = text.replace(" ", "")  # Убрали lower()
    return clean_text == clean_text[::-1]


""" count_char_occurrence мутанты """
# МУТАНТ 4: добавление смещения
def count_char_occurrence_mutant1(text, char):
    return text.count(char) + 1  # Добавлена +1

# МУТАНТ 5: всегда возвращает 0
def count_char_occurrence_mutant2(text, char):
    return 0  # Всегда возвращает 0

# МУТАНТ 6: подсчет только первого символа
def count_char_occurrence_mutant3(text, char):
    return 1 if char in text else 0  # Только факт наличия


""" reverse_string мутанты """
# МУТАНТ 7: возврат оригинала
def reverse_string_mutant1(text):
    return text  # Возвращает оригинал вместо reverse

# МУТАНТ 8: частичный реверс
def reverse_string_mutant2(text):
    return text[-1] + text[1:-1] + text[0] if len(text) > 1 else text  # Только первый и последний

# МУТАНТ 9: двойной реверс
def reverse_string_mutant3(text):
    return text[::-1][::-1]  # Реверс дважды = оригинал
```

```
""" count_words мутанты """
# МУТАНТ 10: вычитание 1
def count_words_mutant1(text):
    words = text.split()
    return len(words) - 1 if words else 0  # Всегда -1

# МУТАНТ 11: подсчет пробелов вместо слов
def count_words_mutant2(text):
    return text.count(' ')  # Считаем пробелы вместо слов

# МУТАНТ 12: всегда возвращает 1
def count_words_mutant3(text):
    return 1 if text.strip() else 0  # Только факт наличия текста

""" count_all_chars мутанты """
# МУТАНТ 13: удвоенное увеличение счетчика
def count_all_chars_mutant1(text):
    char_count = {}
    for character in text:
        if character in char_count:
            char_count[character] += 2  # +2 вместо +1
        else:
            char_count[character] = 1
    return char_count

# МУТАНТ 14: пропуск первого символа
def count_all_chars_mutant2(text):
    char_count = {}
    for character in text[1:]:  # Пропускаем первый символ
        if character in char_count:
            char_count[character] += 1
        else:
            char_count[character] = 1
    return char_count

# МУТАНТ 15: всегда пустой словарь
def count_all_chars_mutant3(text):
    return {}  # Всегда пустой результат
```

Тестирование мутантов показало следующие результаты:

Листинг 4.4. Результат тестирования мутантов

```
ТЕСТИРОВАНИЕ МУТАНТОВ:
 1. ВЫЖИЛ: изменение оператора (!= вместо ==)
 2. ВЫЖИЛ: удаление обработки пробелов
 3. ВЫЖИЛ: удаление lower()
 4. УБИТ: добавлена +1
 5. ВЫЖИЛ: всегда возвращает 0
 6. ВЫЖИЛ: только факт наличия
 7. УБИТ: возвращает оригинал
 8. ВЫЖИЛ: частичный реверс
 9. ВЫЖИЛ: двойной реверс
10. ВЫЖИЛ: вычитание 1
11. УБИТ: подсчет пробелов
12. ВЫЖИЛ: всегда 1 если есть текст
13. УБИТ: удвоенное увеличение
14. ВЫЖИЛ: пропуск первого символа
15. ВЫЖИЛ: всегда пустой словарь
```

Результат показал, что было убито всего 4 мутанта, а покрытие мутационным тестированием составило всего 26.7%.

Улучшим текущий набор тестов, разработав новые для убийства всех мутантов.

Листинг 4.5. Добавленные тесты.

```
# Тесты для is_palindrome
print("assert is_palindrome('топот') == True")
print("assert is_palindrome('Topot') == True  # Регистр")
print("assert is_palindrome('А роза упала на лапу Азора') == True  # Пробелы")

# Тесты для count_char_occurrence
print("assert count_char_occurrence('hello', 'l') == 2  # Точное значение")
print("assert count_char_occurrence('hello', 'x') == 0  # Точное значение")

# Тесты для reverse_string
print("assert reverse_string('hello') == 'olleh'  # Точный реверс")
print("assert reverse_string('hello') != 'hello'  # Не равно оригиналу")

# Тесты для count_words
print("assert count_words('hello world') == 2  # Точное значение")
print("assert count_words('') == 0  # Пустая строка")

# Тесты для count_all_chars
print("assert count_all_chars('hello') == {'h':1, 'e':1, 'l':2, 'o':1}  # Точный словарь")
print("assert count_all_chars('hello') != {}  # Не пустой")
```

После повторного тестирования результат показал, что все мутанты были убиты, и покрытие мутационным тестированием теперь составляет 100%.

Вывод: познакомились с процессом модульного и мутационного тестирования, включая разработку, проведение тестов, исправление ошибок, анализ тестового покрытия, а также оценку эффективности тестов путём применения методов мутационного тестирования.