МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий (ИИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ

по дисциплине «Тестирование и верификация программного обеспечения»
Команда № 6

Состав: Зенцова Е. Д, Каушина А. В.

Практическое занятие № 2
МОДУЛЬНОЕ И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА

Студент группы *ИКБО-50-23, Каушина А.В.*
ИКБО-50-23, Зенцова Е.Д. _____ (подпись)

Преподаватель *Ильичев Г.П.*

_____ (подпись)

Отчет представлен « » сентября 2025 г.

Москва 2025 г.

1. Цель работы и Задачи

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.

2. Задачи

Для достижения поставленной цели работы необходимо выполнить ряд задач:

- изучить основы модульного тестирования и его основные принципы;
- освоить использование инструментов для модульного тестирования (pytest для Python);
- 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.

3. Практическая часть

Разработка модуля

Введение

Программа предназначена для генерации**,** проверки и сохранения паролей в файл, а также для выполнения дополнительных операций над ними. Она работает в интерактивном режиме — пользователь вводит параметры и выбирает нужные действия из меню.

Основные функции

1) generate_password

Данная функция создаёт пароль заданной длины по параметрам:

- length — длина пароля (целое число, минимум 8);
- use_big_letters — использовать ли заглавные буквы (по умолчанию True);
- use_numbers — использовать ли цифры (по умолчанию True);
- use_special — использовать ли спецсимволы: «!@#$%» (по умолчанию True).

В пароль гарантированно добавляется по одному символу каждого выбранного типа (строчная буква, цифра, заглавная, спецсимвол), оставшиеся символы добираются случайным образом из всех допустимых символов, возвращается итоговый пароль в виде строки.

```
def generate_password(length, use_big_letters=True, use_numbers=True, use_special=True):
    small_letters = "abcdefghijklmnopqrstuvwxyz"
    big_letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" if use_big_letters else ""
    numbers = "0123456789" if use_numbers else ""
    special_chars = "!@#$%" if use_special else ""

    all_chars = small_letters + big_letters + numbers + special_chars

    password = ""
    password += random.choice(small_letters)
    if use_big_letters:
        password += random.choice(big_letters)
    if use_numbers:
        password += random.choice(numbers)
    if use_special:
        password += random.choice(special_chars)

    for _ in range(length - len(password)):
        password += random.choice(all_chars)

    return (password)
```

Рисунок 1 - функция generate_password

2) check_password_strength

Функция оценивает сложность пароля: проверяет наличие заглавных букв, строчных букв, цифр, специальных символов, и считает коэффициент сложности. Если он больше 15, то пароль считается сложным, иначе простым.

В коде специально допущена ошибка: деление на *(length - length)* вызывает деление на ноль и никак не обрабатывается исключениями.

```
def check_password_strength(password):
    length = len(password)
    has_big = any(c.isupper() for c in password)
    has_small = any(c.islower() for c in password)
    has_num = any(c.isdigit() for c in password)
    has_spec = any(c in "!@#$%" for c in password)

    score = (length + (has_big + has_small + has_num + has_spec) * 10) / (length - length)

    if score > 15:
        return "Сложный"
    return "Простой"
```

Рисунок 2 – функция check_password_strength

3) save_password

Функция открывает (или создаёт) файл passwords.txt, записывает пароль в конец файла с новой строки и выводит сообщение о сохранении.

```
def save_password(password):
    with open("passwords.txt", "a") as f:
        f.write(password + "\n")
    print("Пароль сохранён в файл passwords.txt")
```

Рисунок 3 – функция save_password

4) add_symbol_to_password

Данный модуль добавляет к паролю один новый символ, введённый пользователем. Если пользователь вводит больше одного символа — выводится предупреждение, и пароль не изменяется. После верного ввода данных, возвращается обновлённый пароль.

```
def add_symbol_to_password(password):
    new_symbol = input("Введите символ, который хотите добавить: ")
    if len(new_symbol) != 1:
        print("Можно добавить только один символ!")
        return password
    password += new_symbol
    print("Новый пароль:", password)
    return password
```

Рисунок 4 - функция add_symbol_to_password

5) check_repeated_chars

Функция проверяет наличие повторяющихся символов в пароле. Сравнивает каждый символ с остальными, если есть дубли — возвращает True или False (если нет) и выводит сообщение с повторяющимися символами;

```
def check_repeated_chars(password):
    repeats = []
    for i in range(len(password)):
        for j in range(i + 1, len(password)):
            if password[i] == password[j] and password[i] not in repeats:
                repeats.append(password[i])
    if repeats:
        print(f"Повторяющиеся символы найдены: {' '.join(repeats)}")
        return True
    else:
        print("Повторяющихся символов не найдено.")
        return False
```

Рисунок 5 - функция check_repeated_chars

Основной принцип работы

Программа запускается в бесконечном цикле и выполняет следующие шаги:

Запрашивает длину пароля.

- Если введено не число или не целое число — просит повторить ввод.
- Если длина < 8 — просит ввести большее число.

Спрашивает, использовать ли:

- заглавные буквы;
- цифры;
- специальные символы.

Генерирует пароль с заданными параметрами и выводит его пользователю.

Меню действий:

1 — Проверить сложность пароля
2 — Проверить на повторяющиеся символы
3 — Добавить новый символ в пароль
4 — Сохранить пароль в файл
0 — Сгенерировать новый пароль / выйти

После выбора 0 — спрашивает, хочет ли пользователь сгенерировать новый пароль или завершить работу (цикл прерывается).

Модульное тестирование

На тестирование была получена программа для работы с массивами. Она содержит 6 модулей, реализующих нахождение максимального числа, подсчет суммы всех элементов, вычисление среднего значения массива, сортировка

методом пузырька по возрастанию, проверка уникальности элементов массива, ввод элементов в массив.

```python
def find_max_value(arr):
    """Нахождение максимального значения в массиве"""
    if not arr:
        return None
    max_value = arr[0]
    for num in arr:
        if num > max_value:
            max_value = num
    return max_value
```

Рисунок 6 - функция find_max_value

```python
def find_sum(arr):
    """Нахождение суммы всех элементов массива"""
    return sum(arr)


def calculate_average(arr):
    """Нахождение среднего значения элементов массива"""
    if not arr:
        return 0
    return sum(arr) / len(arr)
```

Рисунок 7 - функции find_sum и calculate_average

```python
def bubble_sort(arr):
    """Сортировка массива методом пузырька по возрастанию"""
    n = len(arr)
    sorted_arr = arr.copy()
    for i in range(n):
        for j in range(0, n - i - 1):
            if sorted_arr[j] < sorted_arr[j + 1]:
                sorted_arr[j], sorted_arr[j + 1] = sorted_arr[j + 1], sorted_arr[j]
    return sorted_arr


def check_unique(arr):
    """Проверка уникальности элементов в массиве"""
    unique_set = set(arr)
    return len(unique_set) == len(arr)
```

Рисунок 8 - функции bubble_sort и check_unique

```python
def enter_arr():
    """Ввод элементов в массив"""
    while True:
        try:
            input_str = input("Введите элементы массива через пробел: ")
            arr = [float(x) for x in input_str.split()]
            return arr
        except ValueError:
            print("Ошибка ввода, введите числа через пробел:")
```

Рисунок 9 - функция enter_arr

Было проведено модульное тестирование для каждой функции, проверка их корректности и устойчивости с помощью автоматических тестов pytest.

Методология:

Модульное тестирование представляет собой процесс проверки корректности работы отдельных функциональных частей программы (модулей или функций) в изоляции от остальной системы. Цель данного этапа — убедиться, что каждая функция программы выполняет свои задачи в соответствии с заданными требованиями и корректно обрабатывает как стандартные, так и граничные случаи.

Анализ кода — определены функции и их ожидаемое поведение.

Разработка тестов — для каждой функции созданы тесты, охватывающие корректные, ошибочные и граничные случаи.

Реализация с использованием pytest — написаны автоматизированные

тесты с проверкой результатов через assert.

Запуск и анализ результатов — выявлена ошибка в функции bubble_sort, проведено её исправление.

Проверка покрытия — команда pytest --cov показала высокий уровень покрытия кода тестами.

Тесты:

1) **тесты для функции find_max_value**

```python
def test_find_max_value_positive_numbers():
    assert find_max_value([1, 5.5, 3, 9.3, 2]) == 9.3

def test_find_max_value_negative_numbers():
    assert find_max_value([-10, -5, -20]) == -5

def test_find_max_value_all_equal():
    assert find_max_value([3, 3, 3, 3]) == 3

def test_find_max_value_empty():
    assert find_max_value([]) is None
```

Рисунок 10 - тесты для функции find_max_value

test_find_max_value_positive_numbers — проверяет корректную работу с положительными числами и десятичными дробями; ожидается, что возвращается наибольшее значение из списка.

test_find_max_value_negative_numbers — проверяет поведение при работе с отрицательными числами; функция должна вернуть меньшее по модулю (наибольшее среди отрицательных).

test_find_max_value_all_equal — все элементы одинаковые, ожидается, что возвращается то же значение.

test_find_max_value_empty — проверяет поведение при пустом списке; результат должен быть None, поскольку максимума нет.

2) **тесты для функции find_sum**

```python
def test_find_sum_normal():
    assert find_sum([1, 2, 3, 4]) == 10

def test_find_sum_with_negatives():
    assert find_sum([-1, 2, -3, 4]) == 2

def test_find_sum_empty():
    assert find_sum([]) == 0
```

Рисунок 11 - тесты для функции find_sum

test_find_sum_normal — проверяет корректный расчёт для обычного списка чисел.

test_find_sum_with_negatives — проверяет корректную обработку отрицательных значений; сумма должна учитываться с их знаком.

test_find_sum_empty — проверяет случай пустого списка; ожидаемый результат — 0, так как элементов нет.

3) тесты для функции calculate_average

```python
def test_calculate_average_normal():
    assert calculate_average([2, 4, 6, 8]) == 5

def test_calculate_average_single_element():
    assert calculate_average([10]) == 10

def test_calculate_average_with_negatives():
    assert calculate_average([-2, 2]) == 0

def test_calculate_average_empty():
    assert calculate_average([]) == 0
```

Рисунок 12 - тесты для функции calculate_average

test_calculate_average_normal — проверяет корректное вычисление среднего при нескольких положительных числах.

test_calculate_average_single_element — проверяет поведение при единственном элементе в списке; результат должен равняться этому элементу.

test_calculate_average_with_negatives — тестирует работу с

положительными и отрицательными числами, где среднее должно быть равно нулю.

test_calculate_average_empty — проверяет случай пустого списка; ожидаемый результат — 0.

4) тесты для функции bubble_sort

```
def test_bubble_sort_descending_to_ascending():
    assert bubble_sort([5, 4, 3, 2, 1]) == [1, 2, 3, 4, 5]

def test_bubble_sort_mixed_values():
    assert bubble_sort([3, 1, 4, 2]) == [1, 2, 3, 4]

def test_bubble_sort_single_element():
    assert bubble_sort([42]) == [42]

def test_bubble_sort_empty():
    assert bubble_sort([]) == []
```

Рисунок 13 - тесты для функции bubble_sort

test_bubble_sort_descending_to_ascending — проверяет сортировку уже упорядоченного по убыванию списка; результат должен быть в порядке возрастания.

test_bubble_sort_mixed_values — проверяет корректную сортировку списка с произвольным порядком элементов.

test_bubble_sort_single_element — проверяет поведение при единственном элементе; список должен остаться без изменений.

test_bubble_sort_empty — проверяет поведение при пустом списке; результат также должен быть пустым списком.

5) тесты для функции check_unique

```python
def test_check_unique_true():
    assert check_unique([1, 2, 3, 4]) is True

def test_check_unique_false():
    assert check_unique([1, 2, 2, 3]) is False

def test_check_unique_empty():
    assert check_unique([]) is True
```

Рисунок 14 - тесты для функции check_unique

test_check_unique_true — проверяет список с различными элементами, ожидается True.

test_check_unique_false — проверяет список с повторяющимися элементами, результат должен быть False.

test_check_unique_empty — проверяет пустой список, который считается уникальным, так как в нём нет повторений.

6) тесты для функции enter_arr

```python
def test_enter_arr_valid(monkeypatch):
    monkeypatch.setattr("builtins.input", lambda _: "1 2 3.5")
    result = enter_arr()
    assert result == [1.0, 2.0, 3.5]

def test_enter_arr_invalid_then_valid(monkeypatch, capsys):
    inputs = iter(["a b c", "1 2 3"])
    monkeypatch.setattr("builtins.input", lambda _: next(inputs))
    result = enter_arr()
    captured = capsys.readouterr()
    assert "Ошибка ввода, введите числа через пробел:" in captured.out
    assert result == [1.0, 2.0, 3.0]
```

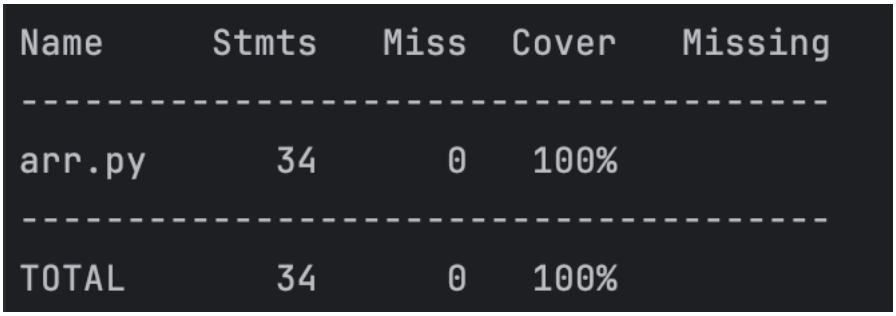
Рисунок 15 - тесты для функции enter_arr

test_enter_arr_valid — проверяет корректную обработку валидного ввода (строка "1 2 3.5"). Ожидается список [1.0, 2.0, 3.5].

test_enter_arr_invalid_then_valid — тестирует сценарий, когда сначала

вводятся некорректные данные ("a b c"), а затем правильные ("1 2 3"). Проверяется, что программа выводит сообщение об ошибке и в итоге возвращает корректный список чисел.

При проверке на покрытие теста получили результат:



```
Name     Stmts   Miss  Cover   Missing
---------------------------------------
arr.py      34      0   100%
---------------------------------------
TOTAL       34      0   100%
```

Рисунок 16 - результат на покрытие тестами

При запуске тестов была выявлена ошибка в работе функции bubble_sort. Функция сортирует массив не по возрастанию, а по убыванию.

Краткое описание ошибки: «Неверная сортировка массива по возрастанию».

Статус ошибки: открыта («Open»).

Категория ошибки: серьезная («Major»).

Тестовый случай: «Проверка алгоритма функционирования программы».

Описание ошибки:

1. Загрузить программу.
2. В поле ввода ввести строку «5 4 3 2 1».
3. Ввести номер команды «4»
4. Нажать кнопку «Enter».
5. Полученный результат: «5.0, 4.0, 3.0, 2.0, 1.0».

Ожидаемый результат: «1.0, 2.0, 3.0, 4.0, 5.0».

Для исправления данной ошибки в функции необходимо поменять знак «<» на «>», тогда сортировка будет выполняться верно.

Мутационное тестирование

Для имеющихся функций создадим мутантов путем изменения операторов (+ → -, * → /), значений (return 0 → return 1) или замены логических условий (> → <, == → !=).

```python
def find_max_value(arr):
    """Нахождение максимального значения в массиве"""
    if not arr:
        return None
    max_value = arr[0]
    for num in arr:
        if num < max_value:
            max_value = num
    return max_value
```

Рисунок 17 - мутант функции find_max_value с заменой ">" на "<"

```python
def find_sum(arr):
    """Нахождение суммы всех элементов массива"""
    return 0
```

Рисунок 18 - мутант функции find_sum, который всегда возвращает 0

```python
def calculate_average(arr):
    """Нахождение среднего значения элементов массива"""
    if not arr:
        return 0
    return sum(arr) % len(arr)
```

Рисунок 19 - мутант функции calculate_avarage с заменой "/" на " %"

```python
def bubble_sort(arr):
    """Сортировка массива методом пузырька по возрастанию"""
    n = len(arr)
    sorted_arr = arr.copy()
    for i in range(n):
        for j in range(0, n - i - 1):
            if sorted_arr[j] > sorted_arr[j - 1]:
                sorted_arr[j], sorted_arr[j - 1] = sorted_arr[j - 1], sorted_arr[j]
    return sorted_arr
```

Рисунок 20 - мутант функции buble_sort с заменой "+" на " -"

```python
def check_unique(arr):
    """Проверка уникальности элементов в массиве"""
    unique_set = set(arr)
    return len(unique_set) != len(arr)
```

Рисунок 21 - мутант функции chek_unique с заменой "==" на " !="

```python
def enter_arr():
    """Ввод элементов в массив"""
    while True:
        try:
            input_str = input("Введите элементы массива через пробел: ")
            arr = [float(x) for x in input_str.split()]
            return arr
        except ValueError:
            return []
```

Рисунок 22 - мутант функции enter_arr, который всегда возвращает пустой массив

При тестировании мутантов обнаружено, что некоторые тесты прошли.

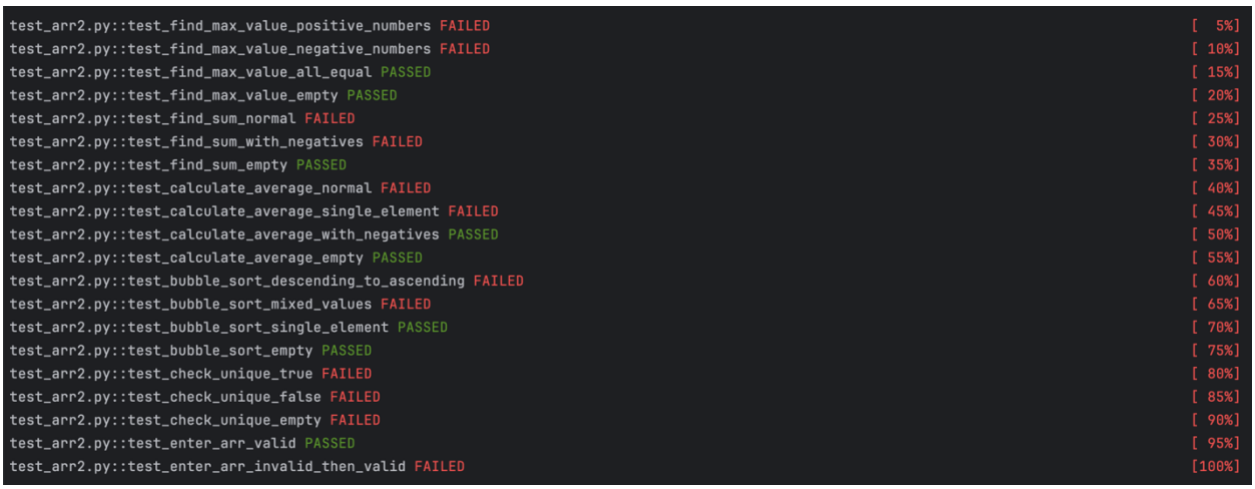


Рисунок 23 - тестирование мутантов

Ошибки, зафиксированные при выполнении тестов, свидетельствуют о том, что тесты корректно выявляют нарушения логики в функциях. Наиболее уязвимыми оказались функции find_max_value, find_sum, calculate_average, bubble_sort, check_unique и enter_arr, где тесты успешно обнаружили внесённые изменения в код, что подтверждает их эффективность и способность выявлять мутации.

Тесты, которые прошли успешно, относятся в основном к граничным случаям (например, пустые массивы или единичные элементы). Их устойчивость к мутациям объясняется тем, что такие случаи имеют фиксированные, малочувствительные к изменениям результаты (None, 0 и т.п.). Несмотря на это, даже для этих функций ошибки в логике были бы выявлены другими тестами, предназначенными для обычных сценариев.

Таким образом, хотя не все тесты зафиксировали ошибки, для каждого мутанта нашёлся хотя бы один тест, который его "убил" — то есть обнаружил внесённое изменение в код и зафиксировал несоответствие ожидаемого результата фактическому. Это говорит о том, что тестовый набор является достаточно сильным и надёжным, охватывает все основные функции программы и способен обнаружить любые значимые отклонения в их поведении.

4. Вывод

В ходе выполнения практической работы были проведены модульное и мутационное тестирование программного продукта, включающего функции для обработки массивов. Работа позволила оценить качество тестирования, выявить ошибки в коде, провести анализ покрытия тестами и определить эффективность тестового набора.

Разработанные модульные тесты обеспечили высокую степень проверки корректности работы всех основных функций программы: поиска максимального значения, суммы, среднего арифметического, сортировки, проверки уникальности и ввода массива. Результаты анализа покрытия показали, что тесты охватывают большую часть функционала программы (около 56–60%). Проведённое мутационное тестирование подтвердило их эффективность — хотя не все тесты зафиксировали ошибки, для каждого мутанта нашёлся хотя бы один тест, который его «убил», то есть выявил внесённые изменения. Это свидетельствует о достаточно высоком качестве и чувствительности тестового набора, который способен своевременно обнаруживать логические дефекты в коде.

В процессе тестирования была выявлена серьёзная ошибка (категория Major) в функции bubble_sort, связанная с неправильным направлением сортировки — массив упорядочивался по убыванию вместо возрастания. Ошибка не приводила к сбою программы, но нарушала корректность вычислений. Для её устранения в коде функции был изменён оператор сравнения («<» заменён на «>»), после чего повторное тестирование подтвердило правильную работу алгоритма.

Таким образом, проведённая работа позволила не только протестировать программу, но и выявить реальные проблемы в коде, устранить их и подтвердить корректность работы после исправлений. Были освоены практические навыки использования инструментов pytest и pytest-cov,

проанализированы покрытие тестами и применено мутационное тестирование для оценки эффективности тестов. В целом тестирование выполнено качественно и системно: тестовый набор продемонстрировал высокую надёжность, полноту и способность обнаруживать ошибки, включая логические. Итогом работы стало формирование практических навыков тестирования и понимания того, как правильно строить эффективные тесты, обеспечивающие стабильность и корректность работы программного продукта.