НЕГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ ЧАСТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «МОСКОВСКИЙ ФИНАНСОВО-ПРОМЫШЛЕННЫЙ УНИВЕРСИТЕТ “СИНЕРГИЯ”»

Факультет/Институт Информационных технологий
(наименование факультета/ Института)

Направление/специальность подготовки: 09.02.07 Информационные системы и программирование
(код и наименование направления /специальности подготовки)

Форма обучения: очная
(очная, очно-заочная, заочная)

Отчет по лабораторной работе № 1

на тему Разработка тестового сценария проекта.
(наименование темы)

по дисциплине Тестирование информационных систем
(наименование дисциплины)

Обучающийся Грачев Дмитрий Александрович _____
(ФИО) (подпись)

Группа ДКИП-312

Преподаватель Авдеенков Владимир Александрович _____
(ФИО) (подпись)

Москва 2024 г

Лабораторная работа №1. «Разработка тестового сценария проекта.»

Задания:

1. Написать (язык – любой) программу решения квадратного уравнения ax2 + bx + c = 0 (задаваемые с клавиатуры коэффициенты – a, b и c);

2. Найти минимальный набор тестов для программы нахождения корней квадратного уравнения ax2 + bx + c = 0 (корни вещественные или комплексные, один/два/бесконечно, один/оба нулевых корня и т. д.);

3. Оформить отчёт (код и скриншоты работы каждого теста, выводы).

1. Код программы на языке Python:

```python
from math import sqrt
from typing import Any


class Discriminant:
    def __init__(self, a: Any, b: Any, c: Any) -> None:
        self.a = self.validation(a)
        self.b = self.validation(b)
        self.c = self.validation(c)


    @staticmethod
    def validation(s: Any) -> Any:
        try:
            return float(s)
        except ValueError:
            return f"Неверный тип данных"


    def multy_valid(self) -> Any:
        if isinstance(self.a, str):
            return self.a
        if isinstance(self.b, str):
            return self.b
        if isinstance(self.c, str):
            return self.c


    def formule(self) -> float:
        return self.b ** 2 - 4 * self.a * self.c


    def two_root(self, d: float) -> tuple:
        x1 = (-self.b + sqrt(d)) / (2 * self.a)
        x2 = (-self.b - sqrt(d)) / (2 * self.a)
        return x1, x2


    def one_root(self) -> float:
        return -self.b / (2 * self.a)


    @staticmethod
    def null_root() -> str:
        return "Нет корней"


    def s_root(self, d: float):
        if d < 0:
            return "Комплексные корни"
        elif d == 0:
            return self.one_root()
        else:
            return self.two_root(d)


    def search_solution(self) -> str:
        if self.a == 0:
            if self.b == 0:
                if self.c == 0:
                    return "Бесконечное число корней"
                return self.null_root()
            return f"Линейное уравнение. Корень: {-self.c / self.b}"
        d = self.formule()
        result = self.s_root(d)
        if isinstance(result, tuple):
            return f"Два корня: x1 = {result[0]}, x2 = {result[1]}"
        return f"Один корень: x = {abs(result) if result == 0 else result}" if d == 0 else result


    def __str__(self) -> str:
        return self.multy_valid() if isinstance(self.multy_valid(), str) else self.search_solution()
```

2. Набор тестов

Сценарий	**a**	**b**	**c**	**Ожидаемый результат**
Ввод коэффициентов с ошибкой	s	2	b	"Неверный тип данных"
Ввод десятичной дроби	0.1	0.2	0	"Два корня: x1 = 0.0, x2 = -2.0"
Комплексные корни	3	2	15	"Комплексные корни"
Нет корней	0	0	1	"Нет корней"
Бесконечное число корней	0	0	0	"Бесконечное число корней"
Нулевой корень	9	0	0	"Один корень: x = 0.0"
Один корень (квадратное уравнение)	1	2	1	"Один корень: x = -1.0"
Один корень (не квадратное уравнение)	0	2	1	"Линейное уравнение. Корень: -0.5"
Два корня	3	-12	0	"Два корня: x1 = 4.0, x2 = 0.0"
Строчное число	"1"	4	"4"	"Один корень: x = -2.0"

Тестирование с использованием pytest на наборе тестов из таблицы

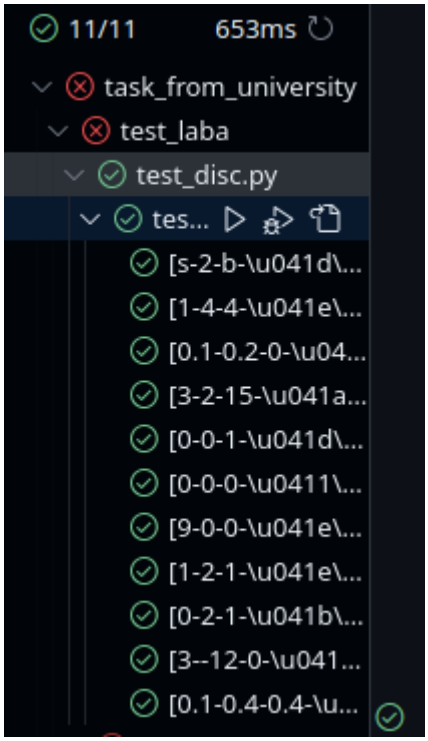
```
from disc import Discriminant
import pytest


test_data = [
    ("s", 2, "b", "Неверный тип данных"),
```

```
    ("1", 4, "4", "Один корень: x = -2.0"),
    (0.1, 0.2, 0, "Два корня: x1 = 0.0, x2 = -2.0"),
    (3, 2, 15, "Комплексные корни"),
    (0, 0, 1, "Нет корней"),
    (0, 0, 0, "Бесконечное число корней"),
    (9, 0, 0, "Один корень: x = 0.0"),
    (1, 2, 1, "Один корень: x = -1.0"),
    (0, 2, 1, "Линейное уравнение. Корень: -0.5"),
    (3, -12, 0, "Два корня: x1 = 4.0, x2 = 0.0"),
    (0.1, 0.4, 0.4, "Один корень: x = -2.0")
]


@pytest.mark.parametrize('a, b, c, expected', test_data)
def test_discriminant(a, b, c, expected):
    assert str(Discriminant(a, b, c)) == expected
```

Результат тестирования – успешное прохождение всех тестов.



Вывод:

В ходе лабораторной работы была написана программа для решения квадратного уравнения по вводимым коэффициентам. Для программы разработан минимальный набор тестов, который проверяет устойчивость программы к наиболее распространенным ошибкам ввода значений, и проверяет алгоритм решения уравнения для различных сценариев.

Теоретические вопросы:

1. Оценка стоимости и причины ошибок в программном обеспечении

Программное обеспечение не идеально и иногда может давать сбои или вести себя неожиданно. Эти сбои или отклонения от запланированной функциональности называются ошибками программного обеспечения. ошибки программного обеспечения могут иметь различные причины, такие как человеческие ошибки, недостатки конструкции, проблемы с оборудованием или внешние факторы. Они также могут иметь различное влияние в зависимости от характера и серьезности ошибки, типа программного обеспечения, а также контекста и среды пользователя.

Ошибки в программном обеспечении могут увеличить расходы и обязательства, связанные с программным обеспечением, как для разработчиков, так и для пользователей. Например, ошибка, из-за которой банковское приложение переводит неправильную сумму денег, может привести к финансовым потерям и юридическим спорам.

Стоимость ошибок в программном обеспечении можно снизить, а качество программного обеспечения можно улучшить, применяя некоторые передовые методы и стратегии на протяжении всего жизненного цикла разработки программного обеспечения. К ним относятся: Планирование и проектирование программного обеспечения с четкими и последовательными требованиями, спецификациями и архитектурой. Это может помочь избежать двусмысленности, путаницы и несогласованности, которые в дальнейшем

могут привести к ошибкам и дефектам. Например, использование гибких методологий, таких как Scrum или Kanban, может облегчить частые и итеративные циклы планирования и обратной связи с заинтересованными сторонами и пользователями.

2. Виды и методы тестирования:

Вид тестирования — это совокупность активностей, направленных на тестирование заданных характеристик системы или её части, основанная на конкретных целях.

Классификация по запуску кода на исполнение:

Статическое тестирование — процесс тестирования, который проводится для верификации практически любого артефакта разработки: программного кода компонент, требований, системных спецификаций, функциональных спецификаций, документов проектирования и архитектуры программных систем и их компонентов.

Динамическое тестирование — тестирование проводится на работающей системе, не может быть осуществлено без запуска программного кода приложения.

Классификация по доступу к коду и архитектуре:

Тестирование белого ящика — метод тестирования ПО, который предполагает полный доступ к коду проекта.

Тестирование серого ящика — метод тестирования ПО, который предполагает частичный доступ к коду проекта (комбинация White Box и Black Box методов).

Тестирование чёрного ящика — метод тестирования ПО, который не предполагает доступа (полного или частичного) к системе. Основывается на работе исключительно с внешним интерфейсом тестируемой системы.

Классификация по уровню детализации приложения:

Модульное тестирование — проводится для тестирования какого-либо одного логически выделенного и изолированного элемента (модуля) системы в коде. Проводится самими разработчиками, так как предполагает полный доступ к коду.

Интеграционное тестирование — тестирование, направленное на проверку корректности взаимодействия нескольких модулей, объединенных в единое целое.

Системное тестирование — процесс тестирования системы, на котором проводится не только функциональное тестирование, но и оценка характеристик качества системы — ее устойчивости, надежности, безопасности и производительности.

Приёмочное тестирование — проверяет соответствие системы потребностям, требованиям и бизнес-процессам пользователя.

Классификация по степени автоматизации:

Ручное тестирование.

Автоматизированное тестирование.

Классификация по принципам работы с приложением

Позитивное тестирование — тестирование, при котором используются только корректные данные.

Негативное тестирование — тестирование приложения, при котором используются некорректные данные и выполняются некорректные операции.

Классификация по уровню функционального тестирования:

Дымовое тестирование (smoke test) — тестирование, выполняемое на новой сборке, с целью подтверждения того, что программное обеспечение стартует и выполняет основные для бизнеса функции.

Тестирование критического пути (critical path) — направлено для проверки функциональности, используемой обычными пользователями во время их повседневной деятельности.

Расширенное тестирование (extended) — направлено на исследование всей заявленной в требованиях функциональности.

Классификация в зависимости от исполнителей:

Альфа-тестирование — является ранней версией программного продукта. Может выполняться внутри организации-разработчика с возможным частичным привлечением конечных пользователей.

Бета-тестирование — программное обеспечение, выпускаемое для ограниченного количества пользователей. Главная цель — получить отзывы клиентов о продукте и внести соответствующие изменения.

Классификация в зависимости от целей тестирования:

Функциональное тестирование (functional testing) — направлено на проверку корректности работы функциональности приложения.

Нефункциональное тестирование (non-functional testing) — тестирование атрибутов компонента или системы, не относящихся к функциональности.

Тестирование производительности (performance testing) — определение стабильности и потребления ресурсов в условиях различных сценариев использования и нагрузок.

Нагрузочное тестирование (load testing) — определение или сбор показателей производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Тестирование масштабируемости (scalability testing) — тестирование, которое измеряет производительность сети или системы, когда количество пользовательских запросов увеличивается или уменьшается.

Объёмное тестирование (volume testing) — это тип тестирования программного обеспечения, которое проводится для тестирования программного приложения с определенным объемом данных.

Стрессовое тестирование (stress testing) — тип тестирования направленный для проверки, как система обращается с нарастающей нагрузкой (количеством одновременных пользователей).

Инсталляционное тестирование (installation testing) — тестирование, направленное на проверку успешной установки и настройки, обновления или удаления приложения.

Тестирование интерфейса (GUI/UI testing) — проверка требований к пользовательскому интерфейсу.

Тестирование удобства использования (usability testing) — это метод тестирования, направленный на установление степени удобства использования, понятности и привлекательности для пользователей разрабатываемого продукта в контексте заданных условий.

Тестирование локализации (localization testing) — проверка адаптации программного обеспечения для определенной аудитории в соответствии с ее культурными особенностями.

Тестирование безопасности (security testing) — это стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.

Тестирование надёжности (reliability testing) — один из видов нефункционального тестирования ПО, целью которого является проверка работоспособности приложения при длительном тестировании с ожидаемым уровнем нагрузки.

Регрессионное тестирование (regression testing) — тестирование уже проверенной ранее функциональности после внесения изменений в код приложения, для уверенности в том, что эти изменения не внесли ошибки в областях, которые не подверглись изменениям.

Повторное/подтверждающее тестирование (re-testing/confirmation testing) — тестирование, во время которого исполняются тестовые сценарии, выявившие ошибки во время последнего запуска, для подтверждения успешности исправления этих ошибок.

3. Понятие теста:

Тест, это инструмент для проверки соответствия программного продукта зада нным требованиям и выявления дефектов.

4. Требования к разработке тестовых сценариев:

Сценарии должны быть четкими и понятными.

Сценарии должны охватывать все возможные варианты использования программного продукта.

Сценарии должны быть независимы друг от друга.

5. Правила разработки тестовых сценариев:

Писать простым и понятным языком без двусмысленных формулировок. Каждый шаг должен быть ясным и однозначным.

Рассматривать не только корректный ход событий, но и варианты нестандартного взаимодействия с приложением. Например, ошибочный ввод данных или нажатие неверных кнопок.

Сохранять готовые сценарии. Некоторые из них могут иметь одинаковые шаги, поэтому не придётся описывать их с нуля.

Вносить в сценарии изменения после каждого обновления функциональности или интерфейса.