**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ: **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА: **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ**

# О Т Ч Е Т

**по лабораторной работе № 2**

**Тема:** Тестирование программного обеспечения (Вариант 11)

**Дисциплина:** Технология разработки программных систем

| | | | |
|---|---|---|---|
| Студент | ИУ6-42Б | 30.05.24 | А. П. Плютто |
| | (Группа) | (Подпись, дата) | (И. О. Фамилия) |
| Преподаватель | | 30.05.24 | Е. К. Пугачёв |
| | | (Подпись, дата) | (И. О. Фамилия) |

Москва, 2024

# Содержание

# 1. Ручное тестирование программных продуктов

## 1.1. Задание

Программа должна формировать типизированный файл с информацией о фамилии человека, дне рождения, а также осуществлять поиск в файле информации о дне рождения (файл исходного кода v11.doc).

## 1.2. Методы ручного тестирования

Основными методами ручного тестирования являются:

- инспекции исходного текста;
- сквозные просмотры;
- просмотры за столом;
- обзоры программ.

Для тестирования методом инспекции исходного текста требуется группа специалистов, в которую входят автор программы, проектировщик,специалист по тестированию и координатор (компетентный программист, но не автор программы), поэтому ручное тестирование данным методом невозможно в рамках данной лабораторной работы.

Для тестирования методом сквозным просмотров требуется группа из 3-5 человек: председатель или координатор, секретарь, фиксирующий все ошибки, специалист по тестированию, программист и независимый эксперт. Поэтотму данный метод тестирования тоже не возможен в рамках выполнения лабораторной работы.

В методе оценки посредством просмотра выбирается программист, который должен выполнять обязанности администратора процесса. Администратор набирает группу от 6 до 20 участников, которые должны быть одного профиля. Каждому участнику предлагается представить для рассмотрения две программы: наилучшую и наихудшую. Отобранные программы случайным образом распределяются между участниками. Им дается по 4 программы — две наилучшие и две наихудшие, но программист не знает, какая из них плохая, а какая хорошая. Программист просматривает их и заполняет анкету, в которой предлагается оценить их относительное качество по шкале из семи баллов. Кроме того, проверяющий дает общий комментарий и рекомендации по улучшению программы. Так как в данной лабораторной требуется оценить только одну программу данный метод тоже не подходит.

## 1.3. Метод проверки за столом

Следующим методом ручного обнаружения ошибок является используемая ранее других методов проверка за столом. Этот метод включает проверку исходного кода программы (или сквозной просмотр), выполняемую одним человеком, который читает код программы, проверяет его по списку вопросов и пропускает через программу тестовые данные. Исходя из принципов тестирования, проверку за столом должен проводить человек, который не является автором программы. Недостатками метода являются:

- полностью неупорядоченный процесс проверки;
- отсутствие обмена мнениями и здоровой конкуренции;
- меньшая эффективность по сравнению с другими методами.

Несмотря на недостатки, данный метод является единственным возможным в рамках выполнения лабораторной работы.

Приведем список вопросов, на которые будем ссылаться позже, в таблице тестирования.

1. Обращения к данным.
    1. Все ли переменные инициализированы?
    2. Не превышены ли максимальные (или реальные) размеры массивов и строк?
    3. Не перепутаны ли строки со столбцами при работе с матрицами?
    4. Присутствуют ли переменные со сходными именами?
    5. Используются ли файлы? Если да, то
        1. при вводе из файла проверяется ли завершение файла?
        2. соответствуют ли типы записываемых и читаемых значений?
    6. Использованы ли нетипизированные переменные, открытые массивы, динамическая память? Если да, то
        1. соответствуют ли типы переменных при наложении формата?
        2. не выходят ли индексы за границы массивов?
2. Вычисления.
    1. Правильно ли записаны выражения (порядок следования операторов)?
    2. Корректно ли производятся вычисления неарифметических переменных?

3. Корректно ли выполнены вычисления с переменными различных типов (в том числе с использованием целочисленной арифметики)?
4. Возможны ли переполнение разрядной сетки или ситуация машинного нуля?
5. Соответствуют ли вычисления заданным требованиям точности?
6. Присутствуют ли сравнения переменных различных типов?

3. Передачи управления.
    1. Будут ли корректно завершены циклы?
    2. Будет ли завершена программа?
    3. Существуют ли циклы, которые не будут выполняться из-за нарушения условия входа? Корректно ли продолжатся вычисления?
    4. Существуют ли поисковые циклы? Корректно ли отрабатываются ситуации «элемент найден» и «элемент не найден»?
4. Интерфейс.
    1. Соответствуют ли списки параметров и аргументов по порядку, типу, единицам измерения?
    2. Не изменяет ли подпрограмма аргументов, которые не должны изменяться?
    3. Не происходит ли нарушения области действия глобальных и локальных переменных с одинаковыми именами?

Далее приведем код непосредственно тестируемой программы.

```cpp
#include <iostream>
#include <string>

using namespace std;

struct fam
{
    string ff;
    short year;
    unsigned char month;
    unsigned char day;
};

int main (){
    FILE* f;
    fam fb;
    int n = 1;
    bool key;
    string fff;

    f = fopen("a.dat", "w");
    cout << "Write data" << endl;

    while (!EOF) {
        cin >> fb.ff >> fb.year >> fb.month >> fb.day;
        fwrite (&fb, sizeof(fb), 1, f);
    };

    fclose (f);

    cout<<"Write lastname"<<endl;
    cin>>fff;

    key = true;
    f = fopen("a.dat", "r");

    while (fread(&fb, sizeof(fb), 1, f) != 20 && key){
        if (fb.ff == fff) {
            cout << "Date:" << fb.year << (short)fb.month
                 << (short)fb.day << endl;
            key = false;
        }
    }

    if (!key)
        cout << "No such lastname" << endl;

    fclose(f);
}

```

Листинг 1 — *Код, предоставленный для ручного тестирования*

Заполним таблицу тестирования данными.

| Номер вопроса | Строки, подлежащие проверке | Результат проверки | Вывод |
|---|---|---|---|
| 1.1 | 15-19, 21, 25, 32, 34 | Переменные, используемые в программе: `f, fb, key, fff` Переменные, инициализированные в программе: `f, fb, n, key, fff` | Все используемые переменные инициализированы |
| 1.2 | 25, 32 | Строки в программе: `fb.ff, fff` | Используются динамические строки типа `std::string`, размеры не могут быть превышены |
| 1.3 | | Работа с матрицами не производится | |
| **1.4** | 25, 32 | Переменные, инициализированные в программе: `f, fb, n, key, fff` | *Переменные со схожими именами присутствуют:* `f, fb, fff, fb.ff` |
| **1.5.1** | 37 | Проверка присутствует, выполнена с помощью сравнения количества данных, прочтенных успешно, с 20 Проверки об успешном открытии файла отсутствуют | *Проверка на завершение при чтении построена некорректно, так как читается всегда только один набор данных* |

| | | | |
|---|---|---|---|
| **1.5.2** | 21, 26, 35, 37 | Файл открыт в режиме `r` и `w`<br>Записываемый тип: `fam`<br>Читаемый тип: `fam` | *Режимы не соответствует условию о типизированных данных. Необходимо открывать файл в бинарных режимах `rb` и `wb`.*<br>Типы соответствуют |
| 1.6 | | Память выделяется динамически только в библеотеках | |
| 2 | | Вычисления в программе не производятся | |
| **3.1** | 24, 37 | Условием завершения первого цикла, вероятнее всего был выбран конец ввода<br>Условием завершения 2 цикла, вероятнее всего, был выбран конец файла | *Оба условия написаны неверно: для конца ввода необходимо проверять не конец файла, а непосредственно возвращаемое значение `cin`, для конца файла можно проверять считываемые значения, но учесть, что всегда считывается одно значение, а не 20* |
| **3.2** | 48-49 | Программа заканчивается закрытием читаемого файла | *Код выхода из программы не возвращен, что может привести к непредвиденным ошибкам после выполнения программы* |

| | | | |
|---|---|---|---|
| 3.3 | 24, 37 | Условиями входа обоих циклов являются открытый на запись и чтение файл | Входы в циклы будут производиться верно |
| 3.4 | 37-46 | Поисковый цикл организован структурно: объявлен флаг выхода если найдено, в условии прописан флаг, после цикла обработка если ничего так и не было найдено | Поисковый алгоритм цикла организован верно |
| **4.1** | 21, 22, 25, 26, 29, 31, 32, 35, 37, 39-40, 46, 48 | Список параметров и аргументов по порядку, типу, единицам измерения соответствует с ожидаемыми параметрами и аргументами библеотечными функциями. Выбор типа `unsigned char` неуместен для месяца и даты, так как преобразование в `short` может выполняться некорректно для некоторых версий языка | *Необходимо изменить тип* `unsigned char` *на другой, к примеру* `short` |
| 4.2 | 6-49 | Структура изменения переменных в программе соответствует условию задачи | Программа не изменяет аргументы, которые не должны изменяться |
| 4.3 | | Глобальные переменные в коде отсутствуют | |

## 1.4. Заключение

При ручном тестировании получили ошибки в 6 пунктах: 1.4, 1.5.1, 1.5.2, 3.1, 3.2, 4.1.

Исправим найденные ошибки и перепишем код:

```cpp
#include <iostream>
#include <string>
#include <cstdio>
#include <sstream>

using namespace std;

struct fam {
    string ff;
    short year;
    short month;
    short day;
};

int main() {
    FILE* f;
    fam fb;
    bool key;
    string fff;

    f = fopen("a.dat", "wb");
    if (f == nullptr) {
        cerr << "Error opening file for writing" << endl;
        return 1;
    }
    cout << "Write data (enter an empty string to stop):" << endl;

    while (true) {
        string input;
        getline(cin, input);
        if (input.empty()) {
            break;
        }
        istringstream iss(input);
        iss >> fb.ff >> fb.year >> fb.month >> fb.day;
        fwrite(&fb, sizeof(fb), 1, f);
    }

    fclose(f);
    cout << "Write lastname:" << endl;
    cin >> fff;
    key = true;
    f = fopen("a.dat", "rb");
    if (f == nullptr) {
        cerr << "Error opening file for reading" << endl;
        return 1;
    }
    while (fread(&fb, sizeof(fb), 1, f) == 1 && key) {
        if (fb.ff == fff) {
            cout << "Date: " << fb.year << " "
                 << fb.month << " "
                 << fb.day << endl;
            key = false;
        }
    }
    if (key) {
        cout << "No such lastname" << endl;
    }
    fclose(f);
    return 0;
}

```

Листинг 2 — *Исправленный, после ручного тестирования, код*

# 2. Белый ящик

## 2.1. Задание

Дана схема алгоритма:

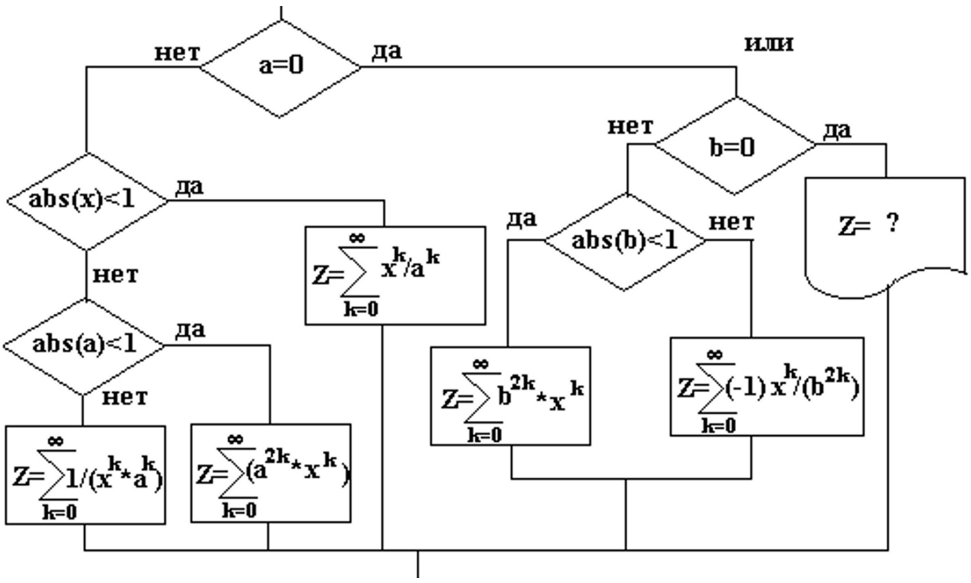


Рисунок 1 — Схема алгоритма

По схеме алгоритма видим, что на вход программы поступают некоторые данные: `a`, `b`, `x`.

## 2.2. Покрытие операторов

Критерий покрытия операторов подразумевает выполнение каждого оператора программы по крайней мере 1 раз. Это необходимое, но недостаточное условие для приемлемого тестирования.

Сделаем таблицу, в которой подберем несколько разных наборов данных, чтобы покрыть все операторы:

| Входные данные | | | Оператор | Z | Программный результат |
|---|---|---|---|---|---|
| a | b | x | | | |
| 1 | $\forall$ | 1 | $Z = \sum_{k=0}^{\infty} \frac{1}{x^k \times a^k}$ | $\infty$ | $\infty$ (число k, до которого программа будет обрабатывать цикл) |
| 0.1 | $\forall$ | 1 | $Z = \sum_{k=0}^{\infty} x^k \times a^{2k}$ | $\frac{100}{99}$ | 1.010101010 |

| | | | | | |
|---|---|---|---|---|---|
| 0.1 | ∀ | 0.1 | $Z = \sum_{k=0}^{\infty} \frac{x^k}{a^k}$ | $\infty$ | $\infty$ (число k, до которого программа будет обрабатывать цикл) |
| 0 | 0.1 | 1 | $Z = \sum_{k=0}^{\infty} x^k \times b^{2k}$ | $\frac{100}{99}$ | 1.010101010 |
| 0 | 1 | 1 | $Z = \sum_{k=0}^{\infty} -\frac{x^k}{b^{2k}}$ | $-\infty$ | $-\infty$ (обратное числу k, до которого программа будет обрабатывать цикл) |

Так как при таком тестировании мы сами подбираем значения нельзя с точностью утверждать, что данное тестирование может покрыть все ошибки, которые могут возникнуть, но так как в результате мы получили ответы, совпадающие с некоторой погрешностью с теоретическими, можно сказать, что данное необходимое условие выполнено.

## 2.3. Покрытие решений

Для реализации этого критерия необходимо достаточное количество тестов, такое, что каждое решение на этих тестах принимает значение «истина» или «ложь» по крайней мере 1 раз

Так как все операторы стоят после условных переходов (т. е. за 1 проход можно попасть только в один из операторов), то при покрытия решений количество тестов, сделанное для покрытия операторов не изменится (так как за один путь мы не можем покрыть 2 оператора).

| Входные данные | | | Назначение теста | Z | Программный результат |
|---|---|---|---|---|---|
| a | b | x | | | |
| 1 | ∀ | 1 | Нет Нет Нет | $\infty$ | $\infty$ (число k, до которого программа будет обрабатывать цикл) |
| 0 | 0.1 | 1 | Да Нет Да | $\frac{100}{99}$ | 1.010101010 |

Минус у данного метод тот же самый: мы не проходим по всему диапазону возможных вариантов, а просто проверяем для нескольких вариантов работу всех условных операторов программы.

## 2.4. Покрытие решений/условий

Этот метод требует составить тесты так, чтобы все возможные результаты каждого условия выполнились по крайней мере 1 раз, все результаты каждого решения выполнились по крайней мере 1 раз и каждой точке входа управление передается по крайней мере 1 раз.

| № | Входные данные | | | Назначение теста | Z | Программный результат |
|---|---|---|---|---|---|---|
| | a | b | x | | | |
| 1 | 1 | ∀ | 1 | Нет Нет Нет | $\infty$ | $\infty$ (число k, до которого программа будет обрабатывать цикл) |
| 2 | 0.1 | ∀ | 1 | Нет Нет Да | $\frac{100}{99}$ | 1.010101010 |
| 3 | 0.1 | ∀ | 0.1 | Нет Да | $\infty$ | $\infty$ (число k, до которого программа будет обрабатывать цикл) |
| 4 | 0 | 0.1 | 1 | Да Нет Да | $\frac{100}{99}$ | 1.010101010 |
| 5 | 0 | 1 | 1 | Да Нет Нет | $-\infty$ | $-\infty$ (обратное числу k, до которого программа будет обрабатывать цикл) |
| 6 | 0 | 0 | ∀ | Да Да | Введенные данные | Введенные данные |

Недостатки метода:

- не всегда можно проверить все условия;
- невозможно проверить условия, которые скрыты другими условиями;
- недостаточная чувствительность к ошибкам в логических выражениях

## 2.5. Комбинаторное покрытие условий

Данный критерий требует создания такого числа тестов, чтобы все возможные комбинации результатов условий в каждом решении и все точки входа выполнялись по крайней мере 1 раз.

Каждая из 3 входных переменных может быть нулем, по модулю меньше 1 (и не ноль) и по модулю больше или равно 1. Обозначим эти состояния переменных цифрами: 0, 1 и 2. Так как у нас 3 переменные, у каждой из которых может быть 3 состояния, то по комбинаторной формуле получаем всего $3 * 3 * 3 = 27$ состояний. Предыдущая таблица была с хорошо проработанными тестами, так что покажем, что все тесты из прошлой таблицы покрывают эти 27 состояний. Получим таблицу:

| Номер теста | Входные данные | | |
|---|---|---|---|
| | a | b | x |
| 1 | 1 | 0 | 1 |
| | 1 | 1 | 1 |
| | 1 | 2 | 1 |
| | 2 | 0 | 1 |
| | 2 | 1 | 1 |
| | 2 | 2 | 1 |
| | 1 | 0 | 0 |
| | 1 | 1 | 0 |
| | 1 | 2 | 0 |
| | 2 | 0 | 0 |
| | 2 | 1 | 0 |
| | 2 | 2 | 0 |

| | | | |
|---|---|---|---|
| 2 | 1 | 0 | 2 |
| | 1 | 1 | 2 |
| | 1 | 2 | 2 |
| 3 | 2 | 0 | 2 |
| | 2 | 1 | 2 |
| | 2 | 2 | 2 |
| 4 | 0 | 0 | 0 |
| | 0 | 0 | 1 |
| | 0 | 0 | 2 |
| 5 | 0 | 1 | 0 |
| | 0 | 1 | 1 |
| | 0 | 1 | 2 |
| 6 | 0 | 2 | 0 |
| | 0 | 2 | 1 |
| | 0 | 2 | 2 |

Данный метод полностью отстраняется от логики и полностью полагается на математическое доказательство, поэтому его достаточно легко реализовать программно и свести вероятность того, что мы не просмотрим все возможные варианты, к нулю.

## 2.6. Заключение

В ходе тестирования было найдено 6 тестов, способных покрыть любые ошибки алгоритма. Арифметических ошибок не было найдено.

## 3. Черный ящик

Одним из способов проверки программ является стратегия тестирования, называемая стратегией «черного ящика» или тестированием с управлением по данным. В этом случае программа рассматривается как «черный ящик» и цель такого тестирования — выяснение обстоятельств, в которых поведение программы не соответствует спецификации. Для обнаружения всех ошибок в программе необходимо выполнить исчерпывающее тестирование, т. е. тестирование на всех возможных наборах данных. Для тех программ, где исполнение команды зависит от предшествующих ей событий, необходимо проверить и все возможные последовательности. Очевидно, что построение исчерпывающего входного теста для большинства случаев невозможно. Поэтому обычно выполняется разумное тестирование, при котором тестирование программы ограничивается прогонами на небольшом подмножестве всех возможных входных данных. Естественно при этом целесообразно выбрать наиболее подходящее подмножество (подмножество с наивысшей вероятностью обнаружения ошибок).

Правильно выбранный тест подмножества должен обладать следующими свойствами:

1. уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования;
2. покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Стратегия «черного ящика» включает в себя следующие методы формирования тестовых наборов:

- эквивалентное разбиение;
- анализ граничных значений;
- анализ причинно-следственных связей;
- предположение об ошибке.

### 3.1. Задание

Программа должна строить график функции по заданным в таблице значениям. Обеспечить возможность выбора вида графика: точки отдельно или точки соединены (исполняемый модуль v11.exe).

Интерфейс программы представлен ниже:


Рисунок 2 — Поле для ввода данных

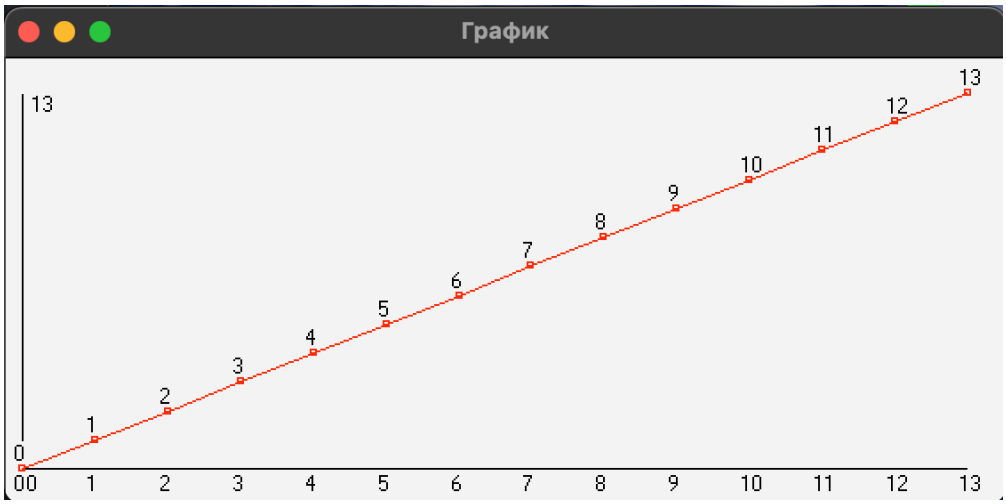


Рисунок 3 — Результат работы программы

## 3.2. Метод эквивалентного разбиения

Основу метода составляют два положения:

1. Исходные данные программы необходимо разбить на конечное число классов эквивалентности, так чтобы можно было предположить, что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если тест какого-либо класса обнаруживает ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку, и наоборот;
2. Каждый тест должен включать по возможности максимальное количество различных входных условий, что позволяет минимизировать общее число необходимых тестов.

Первое положение используется для разработки набора «интересных» условий, которые должны быть протестированы, а второе — для разработки

минимального набора тестов. Разработка тестов методом эквивалентного разбиения осуществляется в два этапа: выделение классов эквивалентности и построение тестов.

Для данной задачи необходимо проверить отрицательные, положительные значения по осям и построение линии относительно предыдущей точки.

То есть максимум для проверки понадобится 2 точки, если программа будет работать для 2 точек, то для большего количества точек она тоже будет работать.

Выделим 5 входных условий:

- Ввод координаты по X точки 1
- Ввод координаты по Y точки 1
- Ввод координаты по X точки 2
- Ввод координаты по Y точки 2
- Отметка соединить точки

| Входное условие | Правильные классы эквивалентности | Неправильные классы эквивалентности |
|---|---|---|
| Ввод координаты по X точки 1 | $x_1 \in (-\infty, \infty)$ | – |
| Ввод координаты по Y точки 1 | $y_1 \in (-\infty, \infty)$ | – |
| Ввод координаты по X точки 2 | $x_2 \in (-\infty, \infty)$ | – |
| Ввод координаты по Y точки 2 | $y_2 \in (-\infty, \infty)$ | – |
| Отметка соединить точки | `true, false` | – |

Так как не на какие координаты не накладываются ограничения по заданию неправильных классов эквивалентности нет.

Составим тесты (тут сразу виден недочет программы: нельзя задать произвольное количество точек, поэтому во время тестирования точки, не участвующие в тестировании будут иметь значения по умолчанию):

| $x_1$ | $y_1$ | $x_2$ | $y_2$ | Соединить точки | Результат |
|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | true | График построен верно |
| 1 | 1 | 2 | 2 | false | График построен верно |
| 2 | 2 | 1 | 1 | true | Ошибка: точка с меньшими координатами не определяется в графике после точки с большими |
| 2 | 0 | 1 | 1 | true | Ошибка: точка с меньшей координатой не определяется в графике после точки с большей |
| 0 | 2 | 1 | 1 | true | Ошибка: точка с меньшей координатой не определяется в графике после точки с большей |
| 1 | 1 | 1 | 1 | true | Ошибка: точка с одинаковыми координатами не определяется в графике после первой точки |
| 0 | 1 | 1 | 1 | true | Ошибка: точка с одинаковой координатой не определяется в графике после первой точки |
| 1 | 0 | 1 | 1 | true | Ошибка: точка с одинаковой координатой не определяется в графике после первой точки |
| −1 | 1 | 2 | 2 | true | График построен верно |
| 1 | −1 | 2 | 2 | true | График построен верно |
| −1 | −1 | 2 | 2 | true | График построен верно |

С помощью тестирования удалось определить 2 ошибки: при вводе точки с меньшими или равными координатами по любой из осей программа перестает работать, второй ошибкой является отсутствие возможности задать другое

количество точек. Данную возможность можно было бы легко добавить, обрабатывая пустые строки отдельно. Так же в процессе тестирования выявлено, что в программе предусмотрено различное кол-во точек: это следует из ошибки о введенной только одной точки, когда программа перестает видеть вторую точку из-за первой ошибки.

## 3.3. Анализ граничных значений

Граничные условия — это ситуации, возникающие вблизи и на границах входных классов эквивалентности.

Анализ граничных значений отличается от эквивалентного разбиения следующим:

- выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных условий осуществляется таким образом, чтобы проверить тестом каждую границу этого класса;
- при разработке тестов рассматриваются не только входные условия (пространство входов), но и пространство результатов.

Анализ граничных условий, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако следует помнить, что граничные условия могут быть едва уловимы и определение их связано с большими трудностями, что является недостатком этого метода.

Второй недостаток связан с тем, что метод анализа граничных условий не позволяет проверять различные сочетания исходных данных.

По условию граничные значения не заданы, поэтому будем проверять все возможные варианты.

После проверки чисел $10^{10}$, $10^{100}$, $10^{200}$, $10^{300}$, $10^{400}$, $10^{310}$, $10^{309}$, $10^{308}$, $10^{307}$ было выявлено, что на отрезке от $10^{307}$ до $10^{308}$ программа ломается и все последующие числа программа просто выдает как что-то, меньшее бесконечности. Так как числа представляются в памяти в двоичном виде логично предположить, что программа ломается как раз на большой степени двойки, $10^{308} \simeq 2^{1024}$. Таким образом можно предположить, что в программе используется динамическая типизация, но с ограничением на выделение памяти до 256 (128 байт = 1024 бит и отриц. числа) байт (или для каждой координаты изначально выделяется 256 байт статически). Поведение программы является неправильным только при очень больших числах, поэтому данные ограничения можно считать приемлемыми.

## 3.4. Анализ причинно-следственных связей

| Назначение теста | Значения исходных данных | Ожидаемый результат | Реакция программы | Вывод |
|---|---|---|---|---|
| Выделение содержимого ячейки для исправления/ удаления | Тройной клик по ячейке | Выделение содержимого | | Программа работает верно |
| Прокручивание горизонтального ползунка | Прокручивание горизонтального ползунка | Прокрутка страницы | | Программа работает верно |
| Блокировка вертикального ползунка | Прокручивание вертикального ползунка | Блокировка действия: пользователю доступны все строки таблицы без прокрутки | Прокрутка таблицы вниз | **Программа работает не верно** |
| Проверка ввода после выделения | Ввод знака и цифр с клавиатуры | Отображение введенного числа в ячейке полностью | Число отображается в ячейке, но не полностью. Доступна прокрутка внутри ячейки | *Программа работает не совсем верно* |
| Проверка работы checkbox’a для соединения точек | Нажатие на checkbox | Изменение значения на противоположное | | Программа работает верно |
| Работоспособность кнопки Построить | Нажатие на кнопку | Отображение графика | | *Программа работает не совсем верно (зависит от входных данных)* |

## 3.5. Предположение об ошибке

Так как в методе эквивалентного разбиения не нашлось никаких условий, ограничивающих точки тесты были сделаны как предположение, где программа может сломаться, после нахождения одной ошибки были составлены тесты, которые проверяли наличие подобных ошибок.

Граничные условия тоже были найдены исходя из факта, что никакой компьютер не способен обработать бесконечность, следовательно подобрав правильное значение можно переполнить даже кучу, затереть стек и вызвать ошибку выполнения.

Так как программа должна строить график при любых двух и более заданных точках, то если эти 2 точки не заданы, необховыводить сообзение об ошибке, иначе просто строить график. Построим логическую схему, которая будет отражать всю работоспособность программы и проверку на то, что любые 2 точки заданы.

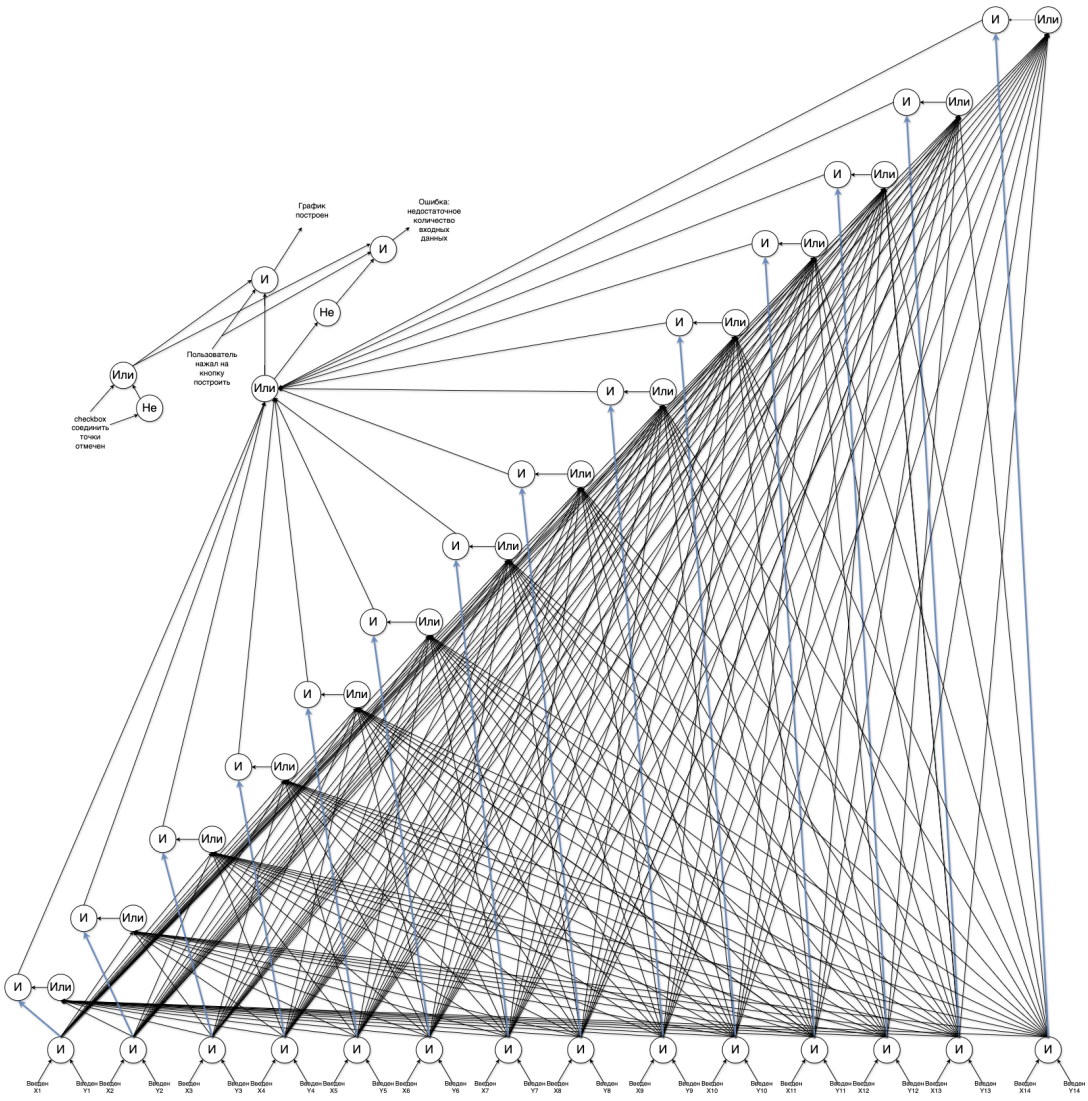


Рисунок 4 — Логическая схема

## 3.6. Заключение

В результате исследования методов тестирования были получены следующие результаты:

- выявлены ошибки в алгоритме построения графика;
- выявлены ошибки в интерфейсе (причинно-следственной связи);
- выявлены ошибки в задании граничных значений;
- оценена специфика каждого метода тестирования;
- оценена трудоемкость тестирования для каждого метода.