МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 «Математика и компьютерные науки»

Лабораторная работа №1
по дисциплине
«Методы тестирования программного обеспечения»

Студент,
группы 5130201/20102 _____ Тищенко А. А.

Преподаватель _____ Курочкин М. А.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение

В разработке ПО, помимо основной задачи — реализовать заявленную в спецификации функциональность, — существует не менее важная задача — обеспечить качество разработанного решения.

Тестирование программного обеспечения — проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов.

На практике ни один метод тестирования не может выявить все ошибки в программе. Это связано с тем, что ресурсы проекта (деньги, время, персонал), в том числе и на тестирование, ограничены. Но все-таки, правильно проведенное тестирование позволяет обнаружить большинство ошибок, что позволит их оперативно исправить, и тем самым повысить качество программного обеспечения.

В данной лабораторной работе используется ручное тестирование — процесс проверки ПО, выполняемый специалистами без использования каких-либо специальных автоматизированных средств.

Ручное тестирование применяется не вместо компьютерного тестирования, а вместе с ним, что позволяет выявить ошибки в программе на более ранних стадиях.

1 Постановка задачи

Требуется: провести инспекцию кода, выступая в роли разработчика программы.

Для выполнения данной задачи необходимо:

- изучить методы ручного тестирования;
- провести инспекцию кода, проанализировать ее результаты;
- исправить программу в соответствии с рекомендациями специалиста по тестированию.

2 Описание методов инспекции кода

Существуют три основных метода ручного тестирования:

- инспекция кода;
- сквозной просмотр;
- тестирование удобства использования.

Эти методы могут применяться на любой стадии разработки ПО, причем как к отдельным готовым модулям или блокам, так и к приложению в целом.

Инспекция и сквозной просмотр включают в себя чтение или визуальную проверку исходного кода программы группой лиц. Оба метода предполагают выполнение определенной подготовительной работы. Завершающим этапом является обмен мнениями между участниками проверки на специальном заседании. Цель такого заседания — нахождение ошибок, но не их устранение (т.е. тестирование, а не отладка). [1]

2.1 Инспекция кода

Инспекция кода — это набор процедур и методик обнаружения ошибок путем анализа (чтения) кода группой специалистов. [1]

2.1.1 Группа инспектирования кода

Обычно в состав группы входят четыре человека, один из которых играет роль координатора. Координатор должен быть квалифицированным программистом, но не автором тестируемой программы, детальное знание которой от него не требуется. Вторым участником группы является программист, а остальными — проектировщик программы (если это не сам программист) и специалист по тестированию. [1]

В рамках выполнения лабораторной работы в инспекции кода участвовало всего три человека: программист, специалист по тестированию и координатор.

2.1.2 Человеческий фактор

Если программист воспринимает инспектирование своей программы как деятельность, направленную против него лично, и занимает оборонительную позицию, то процесс инспектирования не будет эффективным. Программист должен оставить самолюбие в стороне и рассматривать инспекцию только в позитивном и конструктивном ключе, не забывая о том, что целью инспекции является нахождение ошибок и, следовательно, улучшение качества программы. [1]

2.2 Сквозной просмотр

Тестирование программы методом сквозного просмотра также как и в инспекции кода включает в себя проверку программного кода группой лиц.

При сквозном просмотре код проверяется группой разработчиков (оптимально — 3-4 человека), лишь один из которых является автором программы. Таким образом, большую часть программы тестирует не ее создатель, а другие члены команды разработчиков, что согласуется со вторым принципом, согласно которому тестирование программистом собственной программы редко бывает эффективным.

Преимуществом сквозных просмотров, снижающим стоимость отладки (исправления ошибок), является возможность точной локализации ошибки. Кроме того, в процессе сквозного просмотра обычно удается выявить целую группу ошибок, которые впоследствии можно устранить все вместе. При тестировании программы на компьютере обычно проявляются лишь признаки ошибок (например, программа не может корректно завершиться или выводит бессмысленные результаты), а сами они обнаруживаются и устраняются по отдельности. [1]

2.3 Проверка за столом

Метод ручного тестирования «проверка за столом» может рассматриваться как инспекция или сквозной просмотр кода, выполняемые одним человеком, который вычитывает код программы, проверяет его, руководствуясь контрольным списком ошибок, и (или) прогоняет через логику программы тестовые данные.

Для большинства людей проверка за столом является относительно непродуктивной. Это объясняется прежде всего тем, что такая проверка представляет собой полностью неупорядоченный процесс. Вторая, более важная причина заключается в том, что проверка за столом вступает в противоречие со вторым принципом тестирования, согласно которому тестирование программистом собственных программ обычно оказывается неэффективным. Следовательно, оптимальный вариант состоит в том, чтобы такую проверку выполнял человек, не являющийся автором программы (например, два программиста могут обмениваться программами для взаимной проверки, а не проверять собственные программы), но даже в этом случае такая проверка менее эффективна, чем сквозные просмотры или инспекции. В основном именно по этой причине лучше, чтобы сквозные просмотры или инспекции осуществлялись в группе. [1]

2.4 Рецензирование

Рецензирование — это процедура анонимной оценки общих характеристик качества, обслуживаемости, расширяемости, удобства использования и ясности программного обеспечения. Цель данного метода — предоставить программисту возможность получить стороннюю оценку результатов своего труда. Общее руководство процессом осуществляет администратор, выбираемый из числа программистов. В свою очередь, администратор отбирает в группу рецензентов от 6 до 20 участников (6 — это необходимый минимум, обеспечивающий анонимность оценок). Предполагается, что все участники специализируются в одной области. Каждый из участников предоставляет для рецензирования две своих программы, одну из которых он считает наилучшей, а вторую — наихудшей по качеству. Когда будут собраны все программы, их распределяют случайным образом между участниками. Каждому участнику дают для рецензирования четыре программы. Две из них относятся к категории

«наилучших», а две — к категории «наихудших» программ, но рецензенту не сообщают, какой именно является каждая из них. Любой участник тратит на просмотр одной программы 30 минут и заполняет ее оценочную анкету. После просмотра всех четырех программ рецензент оценивает их относительное качество. Рецензента также просят предоставить свои замечания к программе и дать рекомендации по ее улучшению. После просмотра всех программ каждому участнику передают анкеты с оценками двух его программ. Кроме того, участники получают статистическую сводку, отражающую общие и детализированные данные о рейтинге их собственных программ среди всего набора, а также анализ того, насколько оценки, данные участником чужим программам, близки к оценкам тех же программ со стороны других рецензентов [1]

3 Технология инспекции кода

3.1 Состав группы

Заседание происходило в следущем составе:

- секретарь Астафьев Игорь, исполняющий роль координатора;
- специалист по тестированию Кондраев Дмитрий;
- программист Тищенко Артём, он же проектировщик программы.

3.2 Исходный код программы

Код исходного файла программы представлен в листинге 1.

Листинг 1. Исходный код программы `bubble_sort.cpp`.

```
1  #include <iostream>
2  #include <string>
3  #include <locale>
4  #include <vector>
5  #include <windows.h>
6
7  using namespace std;
8
9  struct BSTNode {
10     string word;
11     BSTNode* left;
12     BSTNode* right;
13     BSTNode(const string& w) : word(w), left(nullptr), right(nullptr) {}
14 };
15
16 class BST {
17     BSTNode* root;
18
19     bool insertNode(BSTNode*& node, const string& w) {
20         if (!node) {
```

```
21              node = new BSTNode(w);
22              return true;
23          }
24          if (w == node->word) return false;
25          if (w < node->word) return insertNode(node->left, w);
26          return insertNode(node->right, w);
27      }
28
29      bool removeNode(BSTNode*& node, const string& w) {
30          if (!node) return false;
31          if (w < node->word) return removeNode(node->left, w);
32          if (w > node->word) return removeNode(node->right, w);
33          if (!node->left) {
34              BSTNode* temp = node->right;
35              delete node;
36              node = temp;
37          } else if (!node->right) {
38              BSTNode* temp = node->left;
39              delete node;
40              node = temp;
41          } else {
42              BSTNode* minNode = findMin(node->right);
43              node->word = minNode->word;
44              removeNode(node->right, minNode->word);
45          }
46          return true;
47      }
48
49      BSTNode* findMin(BSTNode* node) {
50          if (!node) return nullptr;
51          while (node->left) node = node->left;
52          return node;
53      }
54
55      void clearTree(BSTNode*& node) {
56          if (!node) return;
57          clearTree(node->left);
58          clearTree(node->right);
59          delete node;
60          node = nullptr;
61      }
62
63      void inorder(BSTNode* node, vector<string>& result) const {
64          if (!node) return;
65          inorder(node->left, result);
66          result.push_back(node->word);
67          inorder(node->right, result);
68      }
69
70  public:
71      BST() : root(nullptr) {}
72      bool insert(const string& w) { return insertNode(root, w); }
73      bool remove(const string& w) { return removeNode(root, w); }
74      void clear() { clearTree(root); }
75      vector<string> getAllWords() const {
76          vector<string> result;
```

```
77          inorder(root, result);
78          return result;
79      }
80  };
81
82  int main() {
83      SetConsoleCP(1251);
84      SetConsoleOutputCP(1251);
85      setlocale(LC_ALL, "Russian");
86      BST dictionary;
87
88      cout << "Команды:\n"
89           << "0 – очистить словарь;\n"
90           << "строка1,<> – добавить строку;\n"
91           << "строка2,<> – удалить строку;\n"
92           << "3 – вывести все слова;\n"
93           << "4 – завершить работу.\n\n";
94
95      while (true) {
96          string input;
97          if (!getline(cin, input)) break;
98          if (input.empty()) {
99              cout << "Некорректный ввод!\n";
100             continue;
101         }
102         int commaPos = input.find(',');
103         int command;
104         string cmdStr, word;
105         if (commaPos == (int)string::npos) {
106             cmdStr = input;
107         } else {
108             cmdStr = input.substr(0, commaPos);
109             if (commaPos + 1 < (int)input.size()) {
110                 word = input.substr(commaPos + 1);
111             }
112         }
113         try { command = stoi(cmdStr); }
114         catch (...) {
115             cout << "Некорректный ввод!\n";
116             continue;
117         }
118
119         switch (command) {
120             case 0:
121                 dictionary.clear();
122                 cout << "Словарь очищен\n";
123                 break;
124             case 1:
125                 if (word.empty()) {
126                     cout << "Некорректный ввод!\n";
127                     break;
128                 }
129                 if (dictionary.insert(word))
130                     cout << "Строка \"" << word << "\" добавлена в словарь\n";
131                 else
132                     cout << "Строка \"" << word << "\" уже есть в словаре\n";
```

```
133             break;
134         case 2:
135             if (word.empty()) {
136                 cout << "Некорректный ввод!\n";
137                 break;
138             }
139             if (dictionary.remove(word))
140                 cout << "Строка \"" << word << "\" удалена из словаря\n";
141             else
142                 cout << "Строка \"" << word << "\" не найдена в словаре\n";
143             break;
144         case 3: {
145             vector<string> allWords = dictionary.getAllWords();
146             if (allWords.empty()) {
147                 cout << "Словарь пуст\n";
148                 break;
149             }
150             for (int i = 0; i < (int)allWords.size(); i++) {
151                 cout << allWords[i];
152                 if (i + 1 < (int)allWords.size()) cout << ", ";
153             }
154             cout << "\n";
155             break;
156         }
157         case 4:
158             return 0;
159         default:
160             cout << "Некорректный ввод!\n";
161             break;
162     }
163 }
164 return 0;
165 }
```

3.3 Список вопросов и ответов на них

Ниже представлен список вопросов, которые специалист по тестированию Кондраев задавал программисту Тищенко. После каждого вопроса приведен ответ. После некоторых ответов следуют замечания специалиста по тестированию.

1. Программа реализует бинарное дерево поиска?

   Ответ: Да.

2. На каком языке написана программа?

   Ответ: C++.

3. Какие типы переменных используются?

   Ответ: int, string, BSTNode*, BST, vector<string>, bool.

4. Все ли константные переменные объявлены при их инициализации?

   Ответ: Да.

5. Где-то используется индексация массивов?

   Ответ: Да, в выводе слов (например, `allWords[i]`) и обработке ввода.

6. Проверяется ли аргумент индексации что он целочисленный?

   Ответ: Да, через приведение типа и проверку условий (например, `(int)allWords.size()`).

7. Корректно ли названы все методы и переменные?

   Ответ: Да, кроме параметра `w` в методах BST.

   Замечание: Параметры вроде `w` стоит переименовать в `word` для ясности.

8. Возможно ли деление на 0?

   Ответ: Нет.

9. Как выполняются проверки сравнения?

   Ответ: Сравниваются строки (например, `w == node->word`) и целые числа.

10. Есть ли операторы сравнения с целыми константами?

    Ответ: Да (например, проверка `command == 0`).

11. Используются ли переменные double, float?

    Ответ: Нет.

12. Может ли какая-нибудь переменная типа int переполниться?

    Ответ: Теоретически возможно, но маловероятно (например, при очень большом количестве слов).

13. Соблюдены ли правила наследования объектов?

    Ответ: Да, наследование не используется.

14. Все ли переменные объявлены?

    Ответ: Да.

15. Правильно ли инициализированы массивы и строки?

    Ответ: Да (например, `vector<string> result;` инициализируется пустым).

16. Удаляются ли неиспользуемые переменные из памяти?

    Ответ: Частично.

    Замечание: Не хватает деструктора для класса `BST` для полной очистки памяти.

17. Присутствуют ли в программе потенциально бесконечные циклы?

    Ответ: Нет.

18. Может ли значение индекса выйти за размер массива?

    Ответ: Нет (используется `(int)allWords.size()`, что предотвращает выход за границы).

19. Проверяются ли входные данные на принадлежность к домену?

    Ответ: Частично.

    Замечание: Не проверяется, содержит ли строка только допустимые символы (например, кириллицу).

20. Используются ли логические выражения?

    Ответ: Да (например, `if (!node)`).

21. Используются ли файлы для ввода-вывода?

    Ответ: Нет.

22. У всех ли классов есть конструктор?

    Ответ: Да (например, `BSTNode(const string& w)`).

23. В цикле for корректно ли написана операция сравнения в цикле?

    Ответ: Да (например, `for (int i = 0; i < (int)allWords.size(); i++)`).

24. Проверяется ли наличие элементов в дереве перед удалением?

    Ответ: Нет.

    Замечание: Метод `remove` возвращает `false` при отсутствии элемента, но нет явной проверки на пустое дерево.

25. Совпадает ли число аргументов, передаваемых вызываемым модулям и число ожидаемых параметров?

    Ответ: Да.

26. Делаются ли в программе попытки поправить входные аргументы?

    Ответ: Да (например, обработка `stoi(cmdStr)` с исключениями).

27. Нет ли пропущенных функций?

    Ответ: Нет, все функции реализованы (вставка, удаление, очистка, вывод).

28. Выдаются ли предупреждения при компиляции?

    Ответ: Нет (при условии корректных настроек компилятора).

29. Выдаются ли ошибки при компиляции?

    Ответ: Нет.

30. Выполняются ли вычисления с присваиванием несовпадающих типов?

    Ответ: Нет.

31. Есть ли комментарии в программе?

    Ответ: Нет.

    Замечание: Стоит добавить пояснения к методам класса `BST`.

32. Оформлен ли код в соответствии с некоторым регламентом?

    Ответ: Да (отступы, именование классов и методов в camelCase).

4 Код программы с внесенными изменениями

Результатом метода инспекции кода является список правок, которые необходимо внести в программу:

1. Переименовать параметр `w` в `word` для улучшения читаемости
2. Добавить деструктор для класса `BST`
3. Добавить проверку входных строк на наличие только буквенных символов
4. Заменить проверки вида `if (!node)` на `if (node == nullptr)`
5. Добавить комментарии к методам класса `BST`

Измененный код программы представлен в листинге 2.

Листинг 2. Модифицированная реализация BST

```
#include <iostream>
#include <string>
#include <locale>
#include <vector>
#include <windows.h>
#include <cctype> // Добавлен новый заголовочный файл

using namespace std;

struct BSTNode {
    string word;
    BSTNode* left;
    BSTNode* right;
    BSTNode(const string& w) : word(w), left(nullptr), right(nullptr) {}
};

class BST {
    BSTNode* root;

    /* Рекурсивная вставка слова в дерево */
    bool insertNode(BSTNode*& node, const string& word) { // Переименован параметр
        if (node == nullptr) { // Изменена проверка на nullptr
            node = new BSTNode(word);
            return true;
        }
        if (word == node->word) return false;
        if (word < node->word) return insertNode(node->left, word);
        return insertNode(node->right, word);
    }

    bool removeNode(BSTNode*& node, const string& word) {
        if (!node) return false;
        if (word < node->word) return removeNode(node->left, word);
        if (word > node->word) return removeNode(node->right, word);
        if (!node->left) {
```

```
36              BSTNode* temp = node->right;
37              delete node;
38              node = temp;
39          } else if (!node->right) {
40              BSTNode* temp = node->left;
41              delete node;
42              node = temp;
43          } else {
44              BSTNode* minNode = findMin(node->right);
45              node->word = minNode->word;
46              removeNode(node->right, minNode->word);
47          }
48          return true;
49      }
50
51      BSTNode* findMin(BSTNode* node) {
52          if (!node) return nullptr;
53          while (node->left) node = node->left;
54          return node;
55      }
56
57      void clearTree(BSTNode*& node) { // Рекурсивная очистка всех узлов дерева
58          if (!node) return;
59          clearTree(node->left);
60          clearTree(node->right);
61          delete node;
62          node = nullptr;
63      }
64
65      void inorder(BSTNode* node, vector<string>& result) const { // Рекурсивный обход дерева в
          порядке возврастания элементов
66          if (!node) return;
67          inorder(node->left, result);
68          result.push_back(node->word);
69          inorder(node->right, result);
70      }
71
72      // Добавлен деструктор
73      ~BST() {
74          clear();
75      }
76
77  public:
78      BST() : root(nullptr) {}
79
80      // Добавлена проверка входных данных
81      bool validateWord(const string& word) {
82          for (char c : word) {
83              if (!isalpha(c) && static_cast<unsigned char>(c) < 128) {
84                  return false;
85              }
86          }
87          return true;
88      }
89
90      bool insert(const string& word) {
```

```
91          if (!validateWord(word)) return false;
92          return insertNode(root, word);
93      }
94
95      bool remove(const string& word) {
96          return removeNode(root, word);
97      }
98
99      void clear() {
100         clearTree(root);
101     }
102
103     vector<string> getAllWords() const {
104         vector<string> result;
105         inorder(root, result);
106         return result;
107     }
108 };
109
110 int main() {
111     // Добавлена проверка кириллических символов
112     SetConsoleCP(1251);
113     SetConsoleOutputCP(1251);
114     setlocale(LC_ALL, "Russian");
115
116     BST dictionary;
117     cout << "Команды:\n"
118          << "0 - очистить словарь;\n"
119          << "строка1,<> - добавить строку;\n"
120          << "строка2,<> - удалить строку;\n"
121          << "3 - вывести все слова;\n"
122          << "4 - завершить работу.\n\n";
123
124     /* Основной цикл обработки ввода */
125     while (true) {
126         string input;
127         if (!getline(cin, input)) break;
128         if (input.empty()) {
129             cout << "Некорректный ввод!\n";
130             continue;
131         }
132         int commaPos = input.find(',');
133         int command;
134         string cmdStr, word;
135         if (commaPos == (int)string::npos) {
136             cmdStr = input;
137         } else {
138             cmdStr = input.substr(0, commaPos);
139             if (commaPos + 1 < (int)input.size()) {
140                 word = input.substr(commaPos + 1);
141             }
142         }
143         try {
144             command = stoi(cmdStr);
145         } catch (...) {
146             cout << "Некорректный ввод!\n";
```

```
147             continue;
148         }
149 
150         switch (command) {
151             case 0:
152                 dictionary.clear();
153                 cout << "Словарь очищен\n";
154                 break;
155             case 1:
156                 if (word.empty() || !dictionary.validateWord(word)) { // Добавлена проверка
157                     cout << "Недопустимые символы в строке!\n";
158                     break;
159                 }
160                 if (dictionary.insert(word))
161                     cout << "Строка \"" << word << "\" добавлена в словарь\n";
162                 else
163                     cout << "Строка \"" << word << "\" уже есть в словаре\n";
164                 break;
165             case 2:
166                 if (word.empty()) {
167                     cout << "Некорректный ввод!\n";
168                     break;
169                 }
170                 if (dictionary.remove(word))
171                     cout << "Строка \"" << word << "\" удалена из словаря\n";
172                 else
173                     cout << "Строка \"" << word << "\" не найдена в словаре\n";
174                 break;
175             case 3: {
176                 vector<string> allWords = dictionary.getAllWords();
177                 if (allWords.empty()) {
178                     cout << "Словарь пуст\n";
179                     break;
180                 }
181                 for (int i = 0; i < (int)allWords.size(); i++) {
182                     cout << allWords[i];
183                     if (i + 1 < (int)allWords.size()) cout << ", ";
184                 }
185                 cout << "\n";
186                 break;
187             }
188             case 4:
189                 return 0;
190             default:
191                 cout << "Некорректный ввод!\n";
192                 break;
193         }
194     }
195     return 0;
196 }
```

Список изменений:

- В строке 21: Переименован параметр `w` в `word`.
- В строке 22: Добавлена явная проверка на nullptr вместо неявного приведения

к bool.

- В строке 73: Добавлен явный деструктор для очистки памяти.
- В строках 81-88: Реализована проверка на допустимые символы (латиница и кириллица).
- В строках 20, 57, 65: Добавлены поясняющие комментарии к методам.

5 Обобщение работы группы

В ходе работы группы в программе были выявлены следующие недостатки: отсутствовала проверка корректности входных данных (например, на допустимые символы в строках); некоторые параметры методов имели неудачные названия (например, параметр `w` вместо `word`); сложные для понимания фрагменты кода, такие как рекурсивные методы работы с деревом, не были снабжены комментариями.

Сделан вывод, что код в целом соответствует стандартам оформления, что делает его удобным для чтения и анализа. Однако для улучшения читаемости и надёжности были предложены и внедрены изменения.

Получен опыт проведения ручного тестирования методом инспекции кода, совместного разбора кода в группе, а также формулирования ответов на вопросы о реализации программы.

Заключение

В ходе выполнения данной лабораторной работы был изучен метод ручного тестирования — инспекция кода. Было проведено инспекционное собрание, на котором участвовали секретарь, программист и специалист по тестированию. По итогу собрания был составлен протокол заседания, далее программистом был проведен анализ составленного протокола и внесены необходимые изменения в код программы. В результате выполненной работы, можно сделать вывод, что методы ручного тестирования, в частности инспекция кода, являются эффективными для выявления ошибок в логике программы, а также для выявления несоответствий требованиям.

В результате работы я пришёл к выводу, что методы ручного тестирования, особенно инспекция кода, эффективны: удалось улучшить читаемость кода, добавить поясняющие комментарии и исправить некоторые недочёты. В частности, было добавлено считывание словаря из файла, выход из бесконечного цикла и обработка пустых строк при вводе для повышения удобства и безопасности.

Список литературы

[1] Майерс, Г. Искусство тестирования программ. – Санкт-Петербург: Диалектика, 2012 г.