Министерство образования и науки
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 1

Инспекция кода

по дисциплине «Методы тестирования программного обеспечения»

Выполнил студент гр. 5130201/10101 _____ Кондраев Дмитрий Евгеньевич
Проверил _____ Курочкин Михаил Александрович

Санкт-Петербург
2024

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 Калугин Евгений, исполняющий роль координатора;
- специалист по тестированию Чалков Кирилл;
- программист Кондраев Дмитрий, он же проектировщик программы.

3.2 Исходный код программы

Task.java

```
import java.io.*;
import java.util.Arrays;
import java.util.Comparator;
import java.util.StringTokenizer;
import java.util.stream.IntStream;

public final class Task {
    private Task() {
        // Should not be instantiated
    }

    private static void solve(final FastScanner in, final PrintWriter out) {
        int nVertices = in.nextInt();
        int mEdges = in.nextInt();
        Graph g = new Graph(nVertices, mEdges);
        for (int i = 0; i < mEdges; i++) {
            g.addEdge(in.nextInt() - 1, in.nextInt() - 1, in.nextInt());
        }
        g.sortEdges();
        out.println(kruskalMstMinMaxPath(g));
    }

    private static int kruskalMstMinMaxPath(Graph g) {
        DisjointSetUnion f = new DisjointSetUnion(g.vertices());
        for (Edge edge : g.edges()) {
            if (f.get(edge.from) == f.get(edge.to)) {
                continue;
            }
            f.union(edge.to, edge.from);
            if (f.get(0) == f.get(g.vertices() - 1)) {
                return edge.weight;
            }
        }
        return Integer.MAX_VALUE;
    }

    private static class DisjointSetUnion {
        final int[] parent;
        final int[] rank;

        public DisjointSetUnion(int size) {
            parent = IntStream.range(0, size).toArray();
            rank = new int[size];
        }
```

```
        public int get(int x) {
            if (x ≠ parent[x]) {
                parent[x] = get(parent[x]);
            }
            return parent[x];
        }

        public void union(int x, int y) {
            x = get(x);
            y = get(y);
            if (rank[x] < rank[y]) {
                parent[x] = y;
            } else {
                parent[y] = x;
            }
            if (rank[x] == rank[y]) {
                rank[y]++;
            }
        }
    }

    private static class Graph {
        private final Edge[] edges;
        private int edgesCount;
        private final int verticesCount;

        public Graph(int verticesCount, int edgesCount) {
            this(verticesCount, new Edge[edgesCount]);
        }

        private Graph(int verticesCount, Edge[] edges) {
            this.edges = edges;
            this.verticesCount = verticesCount;
        }

        public void addEdge(int from, int to, int weight) {
            edges[edgesCount++] = new Edge(from, to, weight);
        }

        public Edge[] edges() {
            return edges;
        }

        public void sortEdges() {
            Arrays.sort(edges, Comparator.comparing(e → e.weight));
        }

        public int vertices() {
            return verticesCount;
        }

        public int edgesCount() {
            return edgesCount;
        }
    }

    private static class Edge {
        public final int from;
        public final int to;
        public final int weight;
```

```
        public Edge(int from, int to, int weight) {
            this.from = from;
            this.to = to;
            this.weight = weight;
        }
    }

    private static class FastScanner {
        private final BufferedReader reader;
        private StringTokenizer tokenizer;

        FastScanner(final InputStream in) {
            reader = new BufferedReader(new InputStreamReader(in));
        }

        String next() {
            while (tokenizer == null || !tokenizer.hasMoreTokens()) {
                try {
                    tokenizer = new StringTokenizer(reader.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return tokenizer.nextToken();
        }

        int nextInt() {
            return Integer.parseInt(next());
        }
    }

    public static void main(final String[] arg) {
        final FastScanner in = new FastScanner(System.in);
        try (PrintWriter out = new PrintWriter(System.out)) {
            solve(in, out);
        }
    }
}
```

3.3 Список вопросов и ответов на них

Ниже представлен список вопросов, которые специалист по тестированию Чалков задавал программисту Кондраеву. После каждого вопроса приведен ответ. После некоторых ответов слудуют замечания специалиста по тестированию.

1. Программа реализует алгоритм Краскала?

   Ответ: Да.

2. На каком языке написана программа?

   Ответ: Java.

3. Какие типы переменных используются?

   Ответ: int, Graph, DisjointSetUnion, Edge.

4. Все ли константные переменные объявлены при их инициализации?

   Ответ: Все переменные инициализируются при объявлении или в конструкторах.

5. Где-то используется индексация массивов?

   Ответ: Да, в методах get, union.

6. Проверяется ли аргумент индексации что он целочисленный?

   Ответ: Да, по типу.

7. Корректно ли названы все методы и переменные?

   Ответ: Да, кроме полей Edge.from, to.

   Замечание: Следует более осмысленно назвать переменные.

8. Возможно ли деление на 0?

   Ответ: Нет.

9. Как выполняются проверки сравнения?

   Ответ: Сравниваются значения, полученные при вызове функций.

10. Есть ли операторы сравнения с целыми константами?

    Ответ: Нет.

11. Используются ли переменные double, float?

    Ответ: Нет.

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

    Ответ: В методе union есть операция ++. Теоретически да, но на практике невозможно, т. к. возможный размер массива в языке ограничен.

13. Соблюдены ли правила наследования объектов?

    Ответ: Наследование не используется.

14. Все ли переменные объявлены?

    Ответ: Да.

15. Правильно ли инициализированы массивы и строки?

    Ответ: Строк в программе нет, а массивы — да.

16. Удаляются ли неиспользуемые переменные из памяти?

    Ответ: Используется автоматический сборщик мусора.

17. Присутствуют ли в программе потенциально бесконечные циклы?

    Ответ: Нет.

18. Может ли значение индекса выйти за размер массива?

    Ответ: Нет.

19. Проверяются ли входные данные на принадлежность к домену?

    Ответ: Не проверяются.

    Замечание: Стоит проверять данные на принадлежность к домену.

20. Используются ли логические выражения?

    Ответ: Да, в методе FastScanner.next.

    Замечание: Тут лучше писать сначала логическое выражение, или константу а потом переменную. Т.к. можно случайно поставить одно = вместо двух.

21. Используются ли файлы для ввода-вывода?

    Ответ: Нет.

22. У всех ли классов есть конструктор?

    Ответ: Да.

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

    Ответ: Да

24. Переменная nEdges всегда больше 0?

    Ответ: На это нет проверки.

    Замечание: Стоит добавить проверку.

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

    Ответ: Да

26. Делаются ли в программе попытки поправить входные аргументы?

    Ответ: В методе DisjointSetUnion.get переменной x присваивается новое значение. В конструкторе Graph — нет.

27. Нет ли пропущенных функций?

    Ответ: Нет. Все функции реализованы.

28. Выдаются ли предупреждения при компиляции?

    Ответ: Нет.

29. Выдаются ли ошибки при компиляции?

    Ответ: Тоже нет.

30. Выполняются ли вычисления с присваиванием несовпадающих типов?

    Ответ: Нет.

31. Есть ли комментарии в программе?

    Ответ: Один есть.

    Замечание: Стоит добавить разъяснительные комментарии.

32. Оформлен ли код в соответствии с некоторым регламентом (стандартом оформления)?

    Ответ: Да.

4 Код программы с внесенными изменениями

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

1. Добавить проверку входных данных на принадлежность к домену.
2. Переписать метод FastScanner.next, изменив порядок условий в логическом выражении.
3. Добавить проверку, что nEdges > 0.
4. Добавить комментарии в программу.
5. Выбрать более осмысленные имена для переменных Edge.from и to.

Task.java

```
import java.io.*;
import java.util.Arrays;
import java.util.Comparator;
import java.util.StringTokenizer;
import java.util.stream.IntStream;

public final class Task {
    private Task() {
        // Should not be instantiated
    }

    private static void solve(final FastScanner in, final PrintWriter out) {
        int nVertices = in.nextInt();
        if (nVertices ≤ 0) {
            out.println("Число вершин не может быть отрицательным или 0.");
            return;
        }
        int mEdges = in.nextInt();
        if (mEdges ≤ 0) {
            out.println("Число ребер не может быть отрицательным или 0.");
            return;
        }
        Graph g = new Graph(nVertices, mEdges);
        for (int i = 0; i < mEdges; i++) {
            g.addEdge(in.nextInt() - 1, in.nextInt() - 1, in.nextInt());
        }
        g.sortEdges();
        out.println(kruskalMstMinMaxPath(g));
    }

    // Алгоритм Краскала поиска минимального остова графа
    // Ребра упорядочены по возрастанию веса.
    // Ребро, на котором останавливается процесс объединения компонент
    // связности --- наибольшего веса в минимальном по весу пути.
    private static int kruskalMstMinMaxPath(Graph g) {
        DisjointSetUnion f = new DisjointSetUnion(g.vertices());
        for (Edge edge : g.edges()) {
            if (f.get(edge.fromVertex) == f.get(edge.toVertex)) {
                continue;
            }
            f.union(edge.toVertex, edge.fromVertex);
            if (f.get(0) == f.get(g.vertices() - 1)) {
                return edge.weight;
            }
```

```
        }
        return Integer.MAX_VALUE;
    }

    // Реализация структуры данных "система непересекающихся множеств".
    // Используется ранговая эвристика на основе глубины деревьев.
    private static class DisjointSetUnion {
        final int[] parent;
        final int[] rank;

        public DisjointSetUnion(int size) {
            parent = IntStream.range(0, size).toArray();
            rank = new int[size];
        }

        public int get(int x) {
            if (x ≠ parent[x]) {
                parent[x] = get(parent[x]);
            }
            return parent[x];
        }

        public void union(int x, int y) {
            x = get(x);
            y = get(y);
            if (rank[x] < rank[y]) {
                parent[x] = y;
            } else {
                parent[y] = x;
            }
            if (rank[x] == rank[y]) {
                rank[y]++;
            }
        }
    }

    private static class Graph {
        private final Edge[] edges;
        private int edgesCount = 0;
        private final int verticesCount;

        public Graph(int verticesCount, int edgesCount) {
            this(verticesCount, new Edge[edgesCount]);
        }

        private Graph(int verticesCount, Edge[] edges) {
            this.edges = edges;
            this.verticesCount = verticesCount;
        }

        public void addEdge(int from, int to, int weight) {
            if (from < 0 || from ≥ verticesCount) {
                throw new IllegalArgumentException("Не является номером вершины: " + from);
            }
            if (to < 0 || to ≥ verticesCount) {
                throw new IllegalArgumentException("Не является номером вершины: " + from);
            }
            edges[edgesCount++] = new Edge(from, to, weight);
        }
```

```
        public Edge[] edges() {
            return edges;
        }

        public void sortEdges() {
            Arrays.sort(edges, Comparator.comparing(e -> e.weight));
        }

        public int vertices() {
            return verticesCount;
        }

        public int edgesCount() {
            return edgesCount;
        }
    }

    private static class Edge {
        public final int fromVertex;
        public final int toVertex;
        public final int weight;

        public Edge(int fromVertex, int toVertex, int weight) {
            this.fromVertex = fromVertex;
            this.toVertex = toVertex;
            this.weight = weight;
        }
    }

    private static class FastScanner {
        private final BufferedReader reader;
        private StringTokenizer tokenizer;

        FastScanner(final InputStream in) {
            reader = new BufferedReader(new InputStreamReader(in));
        }

        String next() {
            while (null == tokenizer || !tokenizer.hasMoreTokens()) {
                try {
                    tokenizer = new StringTokenizer(reader.readLine());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
            return tokenizer.nextToken();
        }

        int nextInt() {
            return Integer.parseInt(next());
        }
    }

    public static void main(final String[] arg) {
        final FastScanner in = new FastScanner(System.in);
        try (PrintWriter out = new PrintWriter(System.out)) {
            solve(in, out);
        }
    }
}
```

5 Обобщение работы группы

В ходе работы группы в программе были найдены некоторые ошибки: не было проверок корректности входных данных; некоторые переменные имели неудачные названия; сложные для понимания фрагменты программы не комментировались.

Сделан вывод, что код писался по регламенту, а значит удобен для чтения.

Получен опыт проведения ручного тестирования методом инспекции кода, разбора кода в группе, ответов на вопросы о коде.

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.

Список использованных источников

1. Майерс, Г. Искусство тестирования программ. – Санкт-Петербург: Диалектика, 2012. – С. 272.