**Министерство образования и науки Российской Федерации**

ФГАОУ ВО «Уральский федеральный университет   
имени первого Президента России Б. Н. Ельцина»

ИРИТ-РТФ

Центр ускоренного обучения

Оценка проекта

Члены комиссии

**РАЗРАБОТКА ПРОГРАММЫ НА JAVAFX ДЛЯ РЕШЕНИЯ ЗАДАЧИ «ЗАДАЧА О КАНАДСКИХ АВИАЛИНИЯХ»**

КУРСОВАЯ РАБОТА

по дисциплине «Объектно-ориентированное программирование»

Пояснительная записка

09.03.04 58.29.29 006 ПЗ

Руководитель В. И. Суханов

Студент гр. РИВ-270027у (662) Э. Ю. Азымов

Екатеринбург 2018

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 3](#_Toc515888864)

[ВВЕДЕНИЕ 4](#_Toc515888865)

[1 Постановка задачи 6](#_Toc515888866)

[2 Анализ поставленной задачи 7](#_Toc515888867)

[2.1 Основные объекты задачи и их взаимодействие 7](#_Toc515888868)

[2.2 Структуры данных 7](#_Toc515888869)

[2.3 Алгоритм 8](#_Toc515888870)

[2.4 Схема работы системы 8](#_Toc515888871)

[3 Описание результатов разработки 10](#_Toc515888872)

[4. Руководство пользователя 30](#_Toc515888884)

[4.1 Минимальные системные требования 30](#_Toc515888885)

[4.2 Установка и удаление программы 30](#_Toc515888886)

[4.3 Использование программы 30](#_Toc515888887)

[ЗАКЛЮЧЕНИЕ 33](#_Toc515888888)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 34](#_Toc515888889)

[ПРИЛОЖЕНИЕ A (справочное) Код программы 35](#_Toc515888890)

# ВВЕДЕНИЕ

Целью курсовой работы является разработка программы, позволяющей составлять маршрут для путешествия по разным городам. Для разработки программы будем использовать язык программирования Java и фреймворк JavaFX. Задача состоит в следующем: дан список городов и их координаты; найти кратчайший маршрут через все города, при этом ни один город нельзя посещать больше одного раза.

Качество написанной программы определяется соблюдением поставленных условий.

Пояснительная записка имеет следующую структуру:

1. во введении описывается краткая постановка задачи и содержание разделов;
2. основная часть состоит из нескольких разделов:
3. постановка задачи – описание требований к написанию курсовой работы;
4. анализ поставленной задачи – взгляд на программу «изнутри», рассмотрение методов решения, предложения по программной реализации (структур данных, алгоритмов);
5. описание результатов разработки – рассказывается о порядке написания программы, в том числе тестирующих модулей. В конце раздела приводится листинг окончательного варианта программы;
6. руководство пользователя – содержит пояснения по использованию программы;
7. в заключении проводится анализ проделанной работы;
8. список использованных источников – содержит перечень использованной при разработке литературы;
9. в приложении содержится руководство пользователя.

# 1 Постановка задачи

Опишем задачу более подробно. Дан список городов и их координаты. Так же есть начальный и конечный города. В процессе путешествия нужно посетить все города по одному разу.

Необходимо разработать программу с графическим интерфейсом, которая составляет маршрут всегда удовлетворяющий вышеуказанным правилам. Ввод/вывод параметров программы осуществляется через графический интерфейс или через файл. Программа должна содержать 4 раздела. Первый раздел – ввод информации и городах и их координатах. Второй раздел – карта на которой отображены города и путь между ними. Третий раздел – ввод начального и конечного города. Четвертый раздел – помощь, информация о пользованием программой. cПрограмма должна быть написана на языке программирования Java и использовать фреймворк JavaFX. Описание результатов разработки оформить в виде пояснительной записки, в соответствии с ГОСТ 2.105-90, 7.32-2001 и 7.1-2003. При описании программы использовать схемы работы системы (ГОСТ 19.701-90) и диаграммы классов по правилам UML.

# 2 Анализ поставленной задачи

## 2.1 Основные объекты задачи и их взаимодействие

Рассмотрим поставленную задачу. Есть сущность «Город», у неё есть следующие свойства: название, координата «X», координата «Y». Из городов формируется список. Вариантом этого списка является кратчайший путь который нужно найти. Так же есть начальный и конечный города.

Каким образом найти кратчайший путь из всех возможных вариантов? Можно воспользоваться методом полного перебора. С помощью вспомогательной функции из заданного списка городов формируется список всех возможных путей, после этого список фильтруется так, чтобы остались только пути начинающиеся с начального города и заканчивающиеся конечным городом. После этого из списка выбирается кратчайший путь путем подсчета стоимости каждого пути.

## 2.2 Структуры данных

Какие структуры данных потребуются при решении задачи? Основная и единственная структура данных – это модель города. Она имеет три поля: строковое поле «name» содержащее название города, целочисленное поле «posX» содержащее координату X, целочисленное поле «posY» содержащее координату Y.

## 2.3 Алгоритм

Определив структуры данных, перейдем к проектированию алгоритма задачи. Одним из вариантов решения задачи может быть следующий:

* 1. инициализация программы с помощью метода main класса Main;
  2. программа, с помощью консоли, запрашивает у пользователя путь к файлу с данными;
  3. с помощью метода readFromFile класса LongestPathFinder программа читает файл и, если файл имеет некорректный формат или некорректные данные, завершается с ошибкой, содержащей описание неполадки. Если данные корректны класс LongestPathFinder заполняет поля содержащие список городов и список рейсов между парами городов. Так же устанавливается город с которого начинается путешествие;
  4. вызвав метод find класса LongestPathFinder, программа получает длиннейший путь для представленного набора городов и рейсов между ними;
  5. длиннейший путь передается методу printPath класса Main. Если длина пути меньше трех, то путь не существует, так как первое и последнее значение это один и тот же город. Пользователю выводится соответствующее сообщение. Если длина пути больше трех, программа выводит путь в консоль.

## 2.4 Схема работы системы

Оформим описанный алгоритм в виде схемы работы системы по ГОСТ 19.701-90 (Рисунок 1).

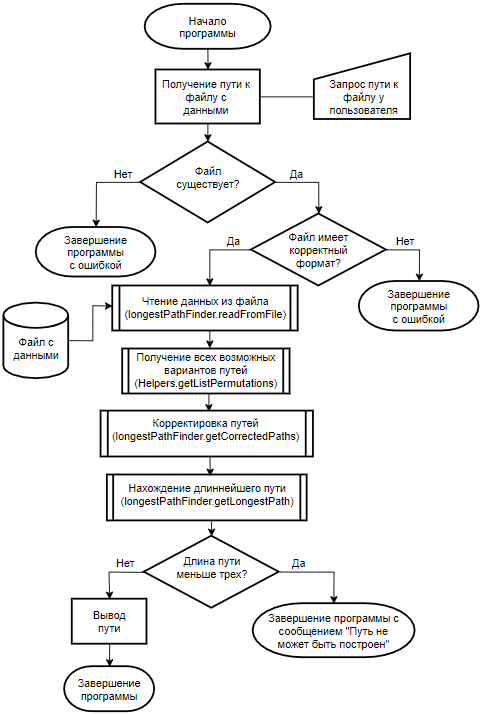


Рисунок 1 – Схема работы системы

# 3 Описание результатов разработки

## 3.1 Первый тест. Класс City

В соответствии с принципами TDD, разработку программы начнем с написания тестов. Для тестирования использовать будем библиотеку JUnit 5.

Первым тестируемым объектом будет класс City. Этот класс должен иметь уникальный идентификатор и наименование которые передаются в конструктор класса при создании экземпляра, для получения этих полей должны быть созданы геттеры getId и getName. Тест должен создавать экземпляр класса с заданными значениями, вызывать геттеры и проверять что возвращаемые ими значения действительно является значениями которые были использованы при создании экземпляра класса.

Исходя из описания требований к тесту, можно выделить три необходимых метода:

1. метод создающий экземпляр класса City с предопределенными значениями и вызывающийся перед каждым другим тестом. Назовем данный метод – setup();
2. метод getId, вызывающий геттер getId экземпляра класса City и проверяющий соответствует ли возвращаемое значение значению, которое использовалось при создании экземпляра;
3. метод getName, вызывающий геттер getName экземпляра класса City и проверяющий соответствует ли возвращаемое значение значению, которое использовалось при создании экземпляра;

Для выполнения тестов воспользуемся аннотациями BeforeEach и Test из библиотеки JUnit. Назовем тестовый класс – CityTest. Так как к предопределенным значениям необходим доступ из всех трех методов, создадим соответствующие переменные класса – id типа String и name типа String.

Изобразим диаграмму класса CityTest, согласно правилам UML (Рисунок 2).

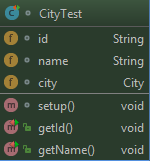


Рисунок 2 – Диаграмма UML класса CityTest

Исходный код класса CityTest приведен в листинге.

package airlines;

import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
class CityTest {  
 String id = "id";  
 String name = "name";  
  
 City city;  
  
 @BeforeEach  
 void setup() {  
 city = new City(id, name);  
 }  
  
 @Test  
 void getId() {  
 assert city.getId() == id;  
 }  
  
 @Test  
 void getName() {  
 assert city.getName() == name;  
 }  
}

Как видно, все методы выполняют описанные выше требования.

## 3.2 Разработка и тестирование класса City

Тест готов, следующим этапом должна стать разработка City. Конструктор класса должен принимать аргумент id типа String и аргумент name типа String. Класс должен иметь геттеры getId и getName. Оформим диаграмму UML разрабатываемого класса (Рисунок 3).

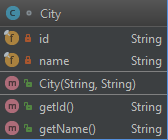


Рисунок 3 – Диаграмма UML класса City

Ориентируясь на диаграмму UML, напишем класс на языке Java.

package airlines;  
  
class City {  
 private final String id;  
 private final String name;  
  
 /\*\*  
 \* @param id уникальный идентификатор города  
 \* @param name Название города  
 \*/  
 public City(String id, String name) {  
 this.id = id;  
 this.name = name;  
 }  
  
 /\*\*  
 \* Возвращает поле id  
 \*  
 \* @return {@code String}  
 \*/  
 public String getId() {  
 return this.id;  
 }  
  
 /\*\*  
 \* Возвращает поле name  
 \*  
 \* @return {@code String}  
 \*/  
 public String getName() {  
 return this.name;  
 }  
}

После запуска теста, мы увидим зеленую полоску – тест пройден!

## 3.3 Второй тест. Класс Ways

Далее протестируем класс Ways, используемый для хранения рейсов между парами городов. Этот класс должен иметь метод addWay для добавления пути и метод hasWay для проверки существования рейса между двумя городами. Тест должен вызывать эти методы и проверять корректно ли они работают.

Исходя из описания требований к тесту, можно выделить три необходимых метода:

1. метод инициализирующий переменные класса. Назовем данный метод – setup();
2. метод addWay, который будет проверять, что метод addWay экземпляра класса Ways добавляет рейс между двумя городами переданными в качестве аргументов;
3. метод hasWay, проверяющий, что метод hasWay экземпляра класса Ways возвращает корректный результат для существующего и несуществующего путей;

Так же необходимо заранее объявить три переменных класса которые будут пересоздаваться перед каждым новым тестом, значениями этих переменных будут экземпляры класса City. Назовем тестовый класс – WaysTest.

Изобразим диаграмму класса WaysTest, согласно правилам UML (Рисунок 4).

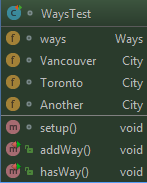


Рисунок 4 – Диаграмма UML класса WaysTest

Класс WaysTest будет выглядеть следующим образом:

package airlines;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
class WaysTest {  
 Ways ways;  
  
 City Vancouver;  
 City Toronto;  
 City Another;  
  
 @BeforeEach  
 void setup() {  
 Vancouver = new City("0", "Vancouver");  
 Toronto = new City("1", "Toronto");  
 Another = new City("2", "Another");  
  
 ways = new Ways();  
 }  
  
 @Test  
 void addWay() {  
 assert ways.hasWay(Vancouver, Toronto) == false;  
  
 ways.addWay(Vancouver.getId(), Toronto.getId());  
  
 assert ways.hasWay(Vancouver, Toronto) == true;  
 }  
  
 @Test  
 void hasWay() {  
 assert ways.hasWay(Vancouver, Toronto) == false;  
  
 ways.addWay(Vancouver.getId(), Toronto.getId());  
  
 assert ways.hasWay(Vancouver, Toronto) == true;  
 assert ways.hasWay(Another, Toronto) == false;  
 }  
}

Как видно, все методы выполняют описанные выше требования.

## 3.4 Разработка и тестирование класса Ways

Тест готов, следующим этапом должна стать разработка класса Ways. Конструктор класса не принимает аргументов. Класс должен иметь приватное поле map для хранения путей, метод addWay для добавления пути между двумя городами, используя их уникальные идентификаторы, и метод hasWay для проверки существования пути между двумя городами. Оформим диаграмму UML разрабатываемого класса (Рисунок 5).

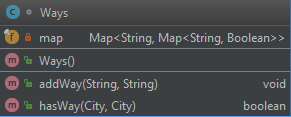


Рисунок 5 – Диаграмма UML класса Ways

Ориентируясь на диаграмму UML, напишем класс на языке Java.

package airlines;  
  
import java.util.HashMap;  
import java.util.Map;  
  
class Ways {  
 private final Map<String, Map<String, Boolean>> map;  
  
 /\*\*  
 \* Инициализирует поле map  
 \*/  
 public Ways() {  
 this.map = new HashMap<>();  
 }  
  
 /\*\*  
 \* Добавляет путь между двумя городами  
 \*  
 \* @param firstCityId уникальный идентификатор первого города  
 \* @param secondCityId уникальный идентификатор второго города  
 \*/  
 public void addWay(String firstCityId, String secondCityId) {  
 if (!map.containsKey(firstCityId)) {  
 map.put(firstCityId, new HashMap<>());  
 }  
  
 if (!map.containsKey(secondCityId)) {  
 map.put(secondCityId, new HashMap<>());  
 }  
  
 map.get(firstCityId).put(secondCityId, true);  
 map.get(secondCityId).put(firstCityId, true);  
 }  
  
 /\*\*  
 \* Проверяет есть ли путь между двумя городами  
 \*  
 \* @param firstCity первый город  
 \* @param secondCity второй город  
 \* @return true если путь между двумя городами существует, иначе false  
 \*/  
 public boolean hasWay(City firstCity, City secondCity) {  
 String firstCityId = firstCity.getId();  
 String secondCityId = secondCity.getId();  
  
 if (!map.containsKey(firstCityId)) return false;  
  
 Map<String, Boolean> firstCityWays = map.get(firstCityId);  
  
 return firstCityWays.containsKey(secondCityId);  
 }  
}

После запуска теста, мы увидим зеленую полоску – тест пройден!

## 3.5 Третий тест. Класс Helpers

Для того, чтобы хранить общие методы не привязанные к какой либо сущности используем класс Helpers. Этот класс должен иметь метод getListPermutations который принимает в качестве аргумента список элементов и возвращает набор всех возможных комбинаций из этих элементов. Тест должен вызывать этот метод с предопределенным списком элементов и проверять, что метод возвращает все возможные наборы этих элементов.

Исходя из описания требований к тесту, можно выделить один метод: getListPermutations который будет вызывать одноименный метод класса Helpers и проверяет корректность возвращаемых значений. Назовем тестовый класс – HelpersTest.

Изобразим диаграмму класса HelpersTest, согласно правилам UML (Рисунок 6).



Рисунок 6 – Диаграмма UML класса HelpersTest

Класс HelpersTest будет выглядеть следующим образом:

package airlines;  
  
import org.junit.jupiter.api.Test;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.assertTrue;  
  
class HelpersTest {  
  
 @Test  
 void getListPermutations() {  
 List<Integer> list = new ArrayList<>();  
  
 list.add(1);  
 list.add(2);  
 list.add(3);  
  
 List<List<Integer>> permutations = Helpers.getListPermutations(list);  
 List<List<Integer>> expected = Arrays.asList(  
 Arrays.asList(1, 2, 3),  
 Arrays.asList(2, 1, 3),  
 Arrays.asList(2, 3, 1),  
 Arrays.asList(1, 3, 2),  
 Arrays.asList(3, 1, 2),  
 Arrays.asList(3, 2, 1)  
 );  
  
 assertTrue(permutations.equals(expected));  
 }  
}

Как видно, все методы выполняют описанные выше требования.

## 3.6 Разработка и тестирование класса Helpers

Тест готов, следующим этапом должна стать разработка класса Helpers. Класс должен иметь один метод – getListPermutations, возвращающий все возможные комбинации из элементов списка подаваемого методу в качестве аргумента. Метод статический поэтому экземпляр класса создавать не нужно. Оформим диаграмму UML разрабатываемого класса (Рисунок 7).



Рисунок 7 – Диаграмма UML класса Helpers

Ориентируясь на диаграмму UML, напишем класс на языке Java.

package airlines;  
  
import java.util.ArrayList;  
import java.util.List;  
  
class Helpers {  
  
 /\*\*  
 \* Возвращает список всех возможных вариаций расстановки элементов списка  
 \*  
 \* @param <T> Тип элемента в массиве  
 \* @param list массив элементов  
 \* @return {@code List<List<T>}  
 \*/  
 public static <T> List<List<T>> getListPermutations(List<T> list) {  
 List<List<T>> result = new ArrayList<>();  
  
 if (list.isEmpty()) {  
 result.add(new ArrayList<>());  
  
 return result;  
 }  
  
 T firstElement = list.remove(0);  
 List<List<T>> permutations = getListPermutations(list);  
  
 permutations.forEach((List<T> smallerPermutated) -> {  
 for (int idx = 0; idx <= smallerPermutated.size(); idx++) {  
 List<T> temp = new ArrayList<>(smallerPermutated);  
  
 temp.add(idx, firstElement);  
 result.add(temp);  
 }  
 });  
  
 return result;  
 }  
}

После запуска теста, мы увидим зеленую полоску – тест пройден!

## 3.7 Четвертый тест. Класс LongestPathFinder

Для произведения операций связанных с вычислением длиннейшего пути используем класс LongestPathFinder. Этот класс должен иметь публичные методы readFromFile для чтения данных из файла и find для вычисления длиннейшего пути исходя из имеющихся данных. Тест должен проверять, что метод readFromFile читает файл и корректно обрабатывает ошибки если файла не существует или он не корректен, а так же проверять, что метод find возвращает список городов.

Исходя из описания требований к тесту, можно выделить три метода:

1. метод инициализирующий создающий новый экземпляр класса LongestPathFinder перед вызовом каждого теста. Назовем данный метод – setup();
2. метод readFromFile, тестирующий одноименный метод класса LongestPathFinder;
3. метод find, тестирующий одноименный метод класса LongestPathFinder.

Назовем тестовый класс – LongestPathFinderTest.

Изобразим диаграмму класса LongestPathFinderTest, согласно правилам UML (Рисунок 8).

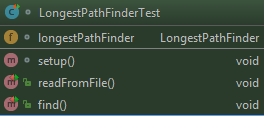


Рисунок 8 – Диаграмма UML класса LongestPathFinderTest

Класс LongestPathFinderTest будет выглядеть следующим образом:

package airlines;  
  
import org.json.simple.parser.ParseException;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class LongestPathFinderTest {  
 LongestPathFinder longestPathFinder;  
  
 @BeforeEach  
 void setup() {  
 longestPathFinder = new LongestPathFinder();  
 }  
  
 @Test  
 void readFromFile() {  
 assertThrows(  
 IllegalArgumentException.class,  
 () -> longestPathFinder.readFromFile("./test/dataWithoutFirstCityId.json")  
 );  
  
 assertThrows(  
 IllegalArgumentException.class,  
 () -> longestPathFinder.readFromFile("./test/dataWithoutCities.json")  
 );  
  
 assertThrows(  
 IllegalArgumentException.class,  
 () -> longestPathFinder.readFromFile("./test/dataWithoutWays.json")  
 );  
 }  
  
 @Test  
 void find() throws IOException, ParseException {  
 City Vancouver = new City("Vancouver", "Vancouver");  
 City Edmonton = new City("Edmonton", "Edmonton");  
 City Winnipeg = new City("Winnipeg", "Winnipeg");  
 ArrayList<City> cities = new ArrayList<>();  
  
 cities.add(Vancouver);  
 cities.add(Edmonton);  
 cities.add(Winnipeg);  
 Ways ways = new Ways();  
  
 ways.addWay(Vancouver.getName(), Edmonton.getName());  
  
 longestPathFinder.setFirstCity(Vancouver);  
 longestPathFinder.setCities(cities);  
 longestPathFinder.setWays(ways);  
  
 List<City> longestPath = longestPathFinder.find();  
 List<City> expected = Arrays.asList(Vancouver, Edmonton, Vancouver);  
  
 assertTrue(longestPath.equals(expected));  
 ways.addWay(Edmonton.getName(), Winnipeg.getName());  
 ways.addWay(Vancouver.getName(), Winnipeg.getName());  
 longestPath = longestPathFinder.find();  
 assertTrue(longestPath.size() == 4);  
 }  
}

## 3.8 Разработка и тестирование класса LongestPathFinder

Тест готов, следующим этапом должна стать разработка класса LongestPathFinder.

Класс должен иметь три приватных поля: cities – для хранения списка городов, firstCity – для хранения первого города и ways – для хранения экземпляра класса Ways, содержащего пути между городами. Для этих трех полей так же должны быть созданы сеттеры.

Конструктору класса не нужно принимать никаких аргументов. Публичный метод readFromFile читает данные из файла по пути, который передается первым аргументом, вызывает три приватных метода, передавая им данные полученные из файла, эти методы заполняют разные переменные класса нужными данными. Публичный метод find производит необходимые калькуляции и возвращает список городов, являющийся длиннейшим путем который был найден. В процессе работы метод вызывает три разных метода:

1. статический метод getListPermutations класса Helpers, получающий все возможные комбинации из заданного списка городов;
2. приватный метод getCorrectPaths класса LongestPathFinder, который получает список путей и отдает корректированный список, каждый путь в котором обрезан на точке разрыва пути;
3. приватный метод getLongestPath класса LongestPathFinder, который перебирает переданные ему пути и возвращает длиннейший из них.

Оформим диаграмму UML разрабатываемого класса (Рисунок 9).

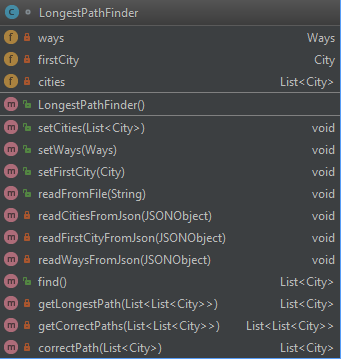


Рисунок 9 – Диаграмма UML класса LongestPathFinder

Ориентируясь на диаграмму UML, напишем класс на языке Java.

package airlines;  
  
import org.json.simple.JSONArray;  
import org.json.simple.JSONObject;  
import org.json.simple.parser.JSONParser;  
import org.json.simple.parser.ParseException;  
  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
  
class LongestPathFinder {  
  
 private Ways ways;  
 private City firstCity;  
 private List<City> cities;  
  
 /\*\*  
 \* Инициализирует значения по умолчанию  
 \*/  
 public LongestPathFinder() {  
 this.cities = new ArrayList<>();  
 this.ways = new Ways();  
 }  
  
 /\*\*  
 \* Задает поле cities  
 \*  
 \* @param cities список городов  
 \*/  
 public void setCities(List<City> cities) {  
 this.cities = cities;  
 }  
  
 /\*\*  
 \* Задает поле waysList  
 \*  
 \* @param waysList набор всех путей между городами  
 \*/  
 public void setWays(Ways waysList) {  
 this.ways = waysList;  
 }  
  
 /\*\*  
 \* Задает поле firstCity  
 \*  
 \* @param firstCity город с которого начинается путешествие  
 \*/  
 public void setFirstCity(City firstCity) {  
 this.firstCity = firstCity;  
 }  
  
 /\*\*  
 \* Читает данные из файла в формате JSON  
 \*  
 \* @param src путь к файлу  
 \* @throws java.io.IOException возбуждается когда файл не найден  
 \* @throws org.json.simple.parser.ParseException возбуждается когда файл имеет некорректный формат  
 \*/  
 public void readFromFile(String src) throws IOException, ParseException {  
 JSONParser parser = new JSONParser();  
 Object text = parser.parse(new FileReader(src));  
 JSONObject json = (JSONObject) text;  
  
 try {  
 readCitiesFromJson(json);  
 readFirstCityFromJson(json);  
 readWaysFromJson(json);  
 } catch (IllegalArgumentException e) {  
 throw new IllegalArgumentException("json не содержит поля " + e.getMessage());  
 }  
 }  
  
 /\*\*  
 \* Читает список городов из JSON  
 \*  
 \* @param json данные в формате JSON которые содержат список городов  
 \*/  
 private void readCitiesFromJson(JSONObject json) {  
 ArrayList<City> citiesList = new ArrayList<>();  
 String jsonCityField = JsonFields.CITIES.toString();  
 JSONArray citiesJson = (JSONArray) json.get(jsonCityField);  
  
 if (citiesJson == null) {  
 throw new IllegalArgumentException(jsonCityField);  
 }  
  
 for (JSONObject cityJson : (Iterable<JSONObject>) citiesJson) {  
 String cityId = (String) cityJson.get("id");  
 String cityName = (String) cityJson.get("name");  
 City city = new City(cityId, cityName);  
  
 citiesList.add(city);  
 }  
  
 this.setCities(citiesList);  
 }  
  
 /\*\*  
 \* Читает уникальный идентификатор первого города из JSON  
 \*  
 \* @param json данные в формате JSON которые содержат ID первого города  
 \*/  
 private void readFirstCityFromJson(JSONObject json) {  
 String jsonFirstCityIdField = JsonFields.FIRST\_CITY\_ID.toString();  
 String firstCityId = (String) json.get(jsonFirstCityIdField);  
  
 if (firstCityId == null) {  
 throw new IllegalArgumentException(jsonFirstCityIdField);  
 }  
  
 for (City city : this.cities) {  
 if (city.getId().equals(firstCityId)) {  
 this.setFirstCity(city);  
 }  
 }  
 }  
  
 /\*\*  
 \* Читает список путей между городами из JSON  
 \*  
 \* @param json данные в формате JSON которые содержат список путей  
 \*/  
 private void readWaysFromJson(JSONObject json) {  
 Ways ways = new Ways();  
 String jsonWaysField = JsonFields.WAYS.toString();  
 JSONArray waysListJson = (JSONArray) json.get(jsonWaysField);  
  
 if (waysListJson == null) {  
 throw new IllegalArgumentException(jsonWaysField);  
 }  
  
 Iterator<JSONArray> waysListIterator = waysListJson.iterator();  
  
 if (waysListJson.isEmpty()) return;  
  
 while (waysListIterator.hasNext()) {  
 JSONArray citiesWayIDs = waysListIterator.next();  
 String firstCityId = (String) citiesWayIDs.get(0);  
 String secondCityId = (String) citiesWayIDs.get(1);  
  
 ways.addWay(firstCityId, secondCityId);  
 }  
  
 this.setWays(ways);  
 }  
  
 /\*\*  
 \* Находит длиннейший возможный путь  
 \*  
 \* @return {@code City<List>}  
 \*/  
 public List<City> find() {  
 List<List<City>> allPaths = Helpers.getListPermutations(cities);  
 List<List<City>> correctPaths = this.getCorrectPaths(allPaths);  
  
 return this.getLongestPath(correctPaths);  
 }  
  
 /\*\*  
 \* Возвращает длиннейший путь из списка переданных  
 \*  
 \* @param paths список путей  
 \* @return {@code List<City>}  
 \*/  
 private List<City> getLongestPath(List<List<City>> paths) {  
 List<City> longestPath = new ArrayList<>();  
  
 for (List<City> curPath : paths) {  
 if (curPath.size() > longestPath.size()) {  
 longestPath = curPath;  
 }  
 }  
  
 return longestPath;  
 }  
  
 /\*\*  
 \* Возвращает отфильтрованный список состоящий из корректных неразрывных  
 \* путей  
 \*  
 \* @param paths корректные пути  
 \* @return {@code List<List<City>>}  
 \*/  
 private List<List<City>> getCorrectPaths(List<List<City>> paths) {  
 List<List<City>> correctPaths = new ArrayList<>();  
  
 for (List<City> path : paths) {  
 if (path.isEmpty()) continue;  
  
 if (path.get(0) != firstCity) continue;  
  
 List<City> correctedPath = this.correctPath(path);  
 City lastCity = correctedPath.get(correctedPath.size() - 1);  
  
 if (ways.hasWay(lastCity, firstCity)) {  
 correctedPath.add(firstCity);  
 correctPaths.add(correctedPath);  
 }  
 }  
  
 return correctPaths;  
 }  
  
 /\*\*  
 \* Возвращает путь обрезанный на точке разрыва  
 \*  
 \* @param path путь  
 \* @return {@code List<City>}  
 \*/  
 private List<City> correctPath(List<City> path) {  
 for (int idx = 1; idx < path.size(); idx++) {  
 City currentCity = path.get(idx);  
 City previousCity = path.get(idx - 1);  
  
 boolean hasWay = this.ways.hasWay(previousCity, currentCity);  
  
 if (!hasWay) {  
 return path.subList(0, idx);  
 }  
 }  
  
 return path;  
 }  
}

После запуска теста, мы увидим зеленую полоску – тест пройден!

## 3.9 Пятый тест. Класс MainTest

Главной точкой входа в программу будет класс Main и его статический метод main. Метод main должен принимать в качестве аргумента массив строк и, если длина этого массива больше нуля, принимать первый элемент массива за путь к файлу, читать данные из этого файла и выводить длиннейший путь исходя из этих данных.

Тест должен вызывать этот метод с предопределенными аргументами и проверять что метод работает с правильно переданными ему аргументами и корректно обрабатывает ошибки.

Исходя из описания требований к тесту, можно выделить три метода:

1. метод setupStreams, который будет вызываться перед каждым тестом и устанавливать наблюдатель за выводом в консоль. Этот наблюдатель нужен чтобы можно было проверить, что программа корректно выводит данные в консоль;
2. метод restoreStreams, который будет вызываться после каждого теста и убирать наблюдатель.
3. метод main, тестирующий одноименный статический метод класса Main.

Назовем тестовый класс – MainTest. Изобразим диаграмму класса MainTest, согласно правилам UML (Рисунок 10).

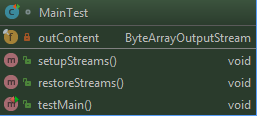


Рисунок 10 – Диаграмма UML класса MainTest

Класс MainTest будет выглядеть следующим образом:

package airlines;  
  
import org.junit.jupiter.api.Test;  
import org.junit.jupiter.api.AfterEach;  
import org.junit.jupiter.api.BeforeEach;  
  
import java.io.PrintStream;  
import java.io.ByteArrayOutputStream;  
  
import static org.junit.jupiter.api.Assertions.assertThrows;  
import static org.junit.jupiter.api.Assertions.assertTrue;  
  
class MainTest {  
 private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();  
  
 @BeforeEach  
 public void setupStreams() {  
 System.setOut(new PrintStream(outContent));  
 }  
  
 @AfterEach  
 public void restoreStreams() {  
 System.setOut(System.out);  
 }  
  
 @Test  
 void main() {  
 String[] ioExceptionArgs = {"NOT A PATH"};  
 String[] parseExceptionArgs = {"./test/wrongFormatData.txt"};  
 String[] rightArgs = {"./test/data.json"};  
 String[] zeroWayArgs = {"./test/dataWithZeroWay.json"};  
  
 assertThrows(  
 IllegalArgumentException.class,  
 () -> Main.main(ioExceptionArgs)  
 );  
  
 assertThrows(  
 IllegalArgumentException.class,  
 () -> Main.main(parseExceptionArgs)  
 );  
 Main.main(rightArgs);  
 assertTrue(outContent.toString().length() > 0);  
  
 outContent.reset();  
  
 Main.main(zeroWayArgs);  
 assertTrue(outContent.toString().length() > 0);  
 }  
}

## 3.10 Разработка и тестирование класса Main

Тест готов, следующим этапом должна стать разработка класса Main.

Класс имеет статический метод main который первым вызывается в начале работы программы. В процессе работы метод проверяет, передан ли путь к файлу с данными через аргументы метода, если нет, то программа запрашивает путь к файлу у пользователя. По указанному пути загружаются данные из файла с помощью метода readFromFile класса LongestPathFinder. С помощью того же класса и метода find получается длиннейший путь. Если длина пути меньше трех, то построение пути невозможно, так как первая и последняя точка пути это один и тот же город. В противном случае путь выводится в консоль с помощью статического метода printPath класса Main. На этом работа программы успешно закончена.

Оформим диаграмму UML разрабатываемого класса (Рисунок 11).

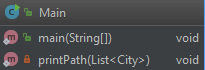


Рисунок 11 – Диаграмма UML класса Main

Ориентируясь на диаграмму UML, напишем класс на языке Java.

package airlines;  
  
import org.json.simple.parser.ParseException;  
  
import java.io.IOException;  
import java.util.List;  
import java.util.Scanner;  
  
public class Main {  
  
 /\*\*  
 \* Получает путь к файлу с данными и выводит длиннейший путь исходя  
 \* из этих данных  
 \*  
 \* @param args аргументы программы  
 \*/  
 public static void main(String[] args) {  
 String src;  
  
 if (args.length == 0) {  
 Scanner scanner = new Scanner(System.in);  
  
 System.out.println("Введите путь до json файла с данными");  
 src = scanner.nextLine();  
 } else {  
 src = args[0];  
 }  
  
 try {  
 LongestPathFinder longestPathFinder = new LongestPathFinder();  
 longestPathFinder.readFromFile(src);  
 List<City> longestPath = longestPathFinder.find();  
  
 printPath(longestPath);  
  
 } catch (IOException e) {  
 throw new IllegalArgumentException("Файл не существует");  
  
 } catch (ParseException e) {  
 throw new IllegalArgumentException("Файл имеет некорректный формат");  
 }  
  
 System.out.println("Нажмите Enter для выхода из программы.");  
  
 try {  
 System.in.read();  
 } catch (Exception ex) {}  
 }  
  
 /\*\*  
 \* Выводит путь  
 \*  
 \* @param path список городов  
 \*/  
 private static void printPath(List<City> path) {  
 if (path.size() < 3) {  
 System.out.println("Путь не может быть построен");  
 return;  
 }  
  
 City firstCity = path.get(0);  
 System.out.println("Начинаем путь с города " + firstCity.getName());  
  
 for (int idx = 1; idx < path.size() - 1; idx++) {  
 System.out.println("-> Летим в " + path.get(idx).getName());  
 }  
  
 System.out.println("Возвращаемся в " + firstCity.getName());  
 }  
}

После запуска теста, мы увидим зеленую полоску – тест пройден!

## 3.11 Общая схема системы

Общая схема системы, она же диаграмма классов, в формате UML представлена ниже (Рисунок 12).

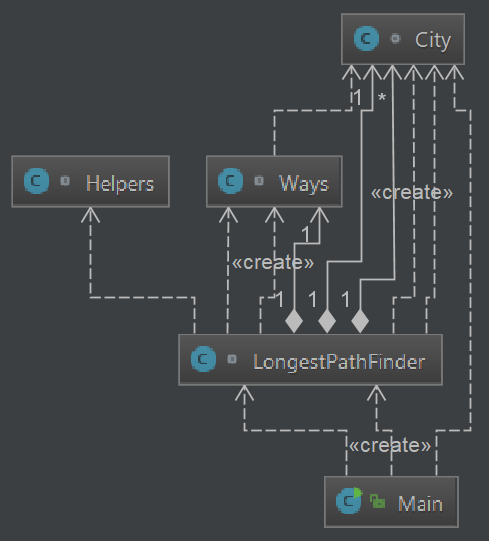


Рисунок 12 – Диаграмма классов

# 4. Руководство пользователя

## 4.1 Минимальные системные требования

Для установки и запуска программы необходимо выполнение следующих минимальных требований:

1. операционная система MS Windows (любая версия);
2. установленная виртуальная машина Java (Java 2 Platform, Standard Edition), не ниже пятой версии. Установочные файлы Java можно загрузить с сервера компании Sun, по адресу: http://java.sun.com.

## 4.2 Установка и удаление программы

Программа поставляется исполняемым файлом App.exe. Пользователь может поместить файл в любую папку и вызывать оттуда. Для установки программы не требуется никаких дополнительных действий.

Чтобы запустить программу нужно дважды кликнуть мышкой по файлу App.exe.

Для удаления программы нужно удалить файл App.exe.

## 4.3 Использование программы

При запуске программы пользователь увидит консольное приложение (Рисунок 13).

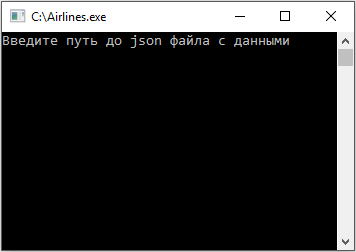


Рисунок 13 – Консольное приложение

Программа предлагает ввести путь к файлу с данными. Путь может быть как абсолютным так и относительным. Ниже на рисунке 14 можно увидеть какой путь нужно вводить если файл с данными называется «data.json» и находится рядом с файлом программы.

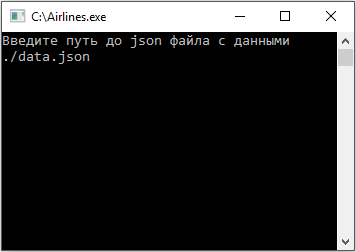


Рисунок 14 – Пример входных данных

Указав файл и нажав клавишу Enter мы увидим ответ программы, сформированный исходя из заданных файлом данных (Рисунок 15).

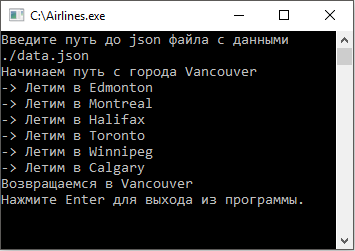


Рисунок 15 – Результат работы программы

Для того чтобы закрыть программу нужно нажать клавишу Enter

Пример формата данных, требуемого программой от файла с данными, можно увидеть в листинге ниже.

{  
 "firstCityId": "Vancouver",  
 "cities": [  
 {  
 "id": "Vancouver",  
 "name": "Vancouver"  
 },  
 {  
 "id": "Yellowknife",  
 "name": "Yellowknife"  
 },  
 {  
 "id": "Edmonton",  
 "name": "Edmonton"  
 },  
 {  
 "id": "Calgary",  
 "name": "Calgary"  
 },  
 {  
 "id": "Winnipeg",  
 "name": "Winnipeg"  
 },  
 {  
 "id": "Toronto",  
 "name": "Toronto"  
 },  
 {  
 "id": "Montreal",  
 "name": "Montreal"  
 },  
 {  
 "id": "Halifax",  
 "name": "Halifax"  
 }  
 ],  
 "ways": [  
 ["Vancouver", "Edmonton"],  
 ["Vancouver", "Calgary"],  
 ["Calgary", "Winnipeg"],  
 ["Winnipeg", "Toronto"],  
 ["Toronto", "Halifax"],  
 ["Montreal", "Halifax"],  
 ["Edmonton", "Montreal"],  
 ["Edmonton", "Yellowknife"],  
 ["Edmonton", "Calgary"]  
 ]  
}

# ЗАКЛЮЧЕНИЕ

В процессе выполнения курсовой работы получены навыки по работе с языком программирования Java и фреймворком JavaFX. Разработано решение задачи «задача о канадских авиалиниях». Составлена схема работы системы, для каждого класса сформированы схемы UML. Составлено руководство пользователя, включающее в себя системные требования, описание процесса по установке и удалению программы, а так же по ее использованию. Приложение A содержит полный текст программы.

Окончательная версия программы – консольное Windows приложение для Java платформы. В ходе выполнения, программа использует файловый ввод и ввод через графический интерфейс.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Окулов С.М. Программирование в алгоритмах [Текст] / С. М. Окулов. – М. : Бином. Лаборатория знаний, 2002. – 341 с.
2. Документация по языку программирования Java [Электронный ресурс] // Сайт Oracle. – [Б. м], [2018]. – Режим доступа: https://docs.oracle.com/javase/8/docs/api/
3. Документация по языку фреймворку JavaFX [Электронный ресурс] // Сайт Oracle. – [Б. м], [2018]. – Режим доступа: https://docs.oracle.com/javafx/2/
4. Ноутон П. JavaTM 2 [Текст]: пер. с англ. / П. Ноутон, Г. Шилдт. – СПб.: БХВ-Петербург, 2005. – 1072 с.
5. Кичигин В.Н. Оформление курсовых и дипломных проектов [Текст]: методические указания для студентов технических специальностей / В. Н. Кичигин, И. Е. Мясников, С. И. Тимошенко. – Екатеринбург: ГОУ ВПО «УГТУ-УПИ», 2005. – 80 с.
6. Тимошенко С. И. Порядок выполнения и требования к оформлению курсовой работы по дисциплине «Программирование на языках высокого уровня» [Текст]: методи-ческие указания / С. И. Тимошенко. – Екатеринбург: изд. ИПК УГТУ, 2004. – 16 с.

# **ПРИЛОЖЕНИЕ A** (справочное) Код программы

## A.1 Класс City

package airlines;