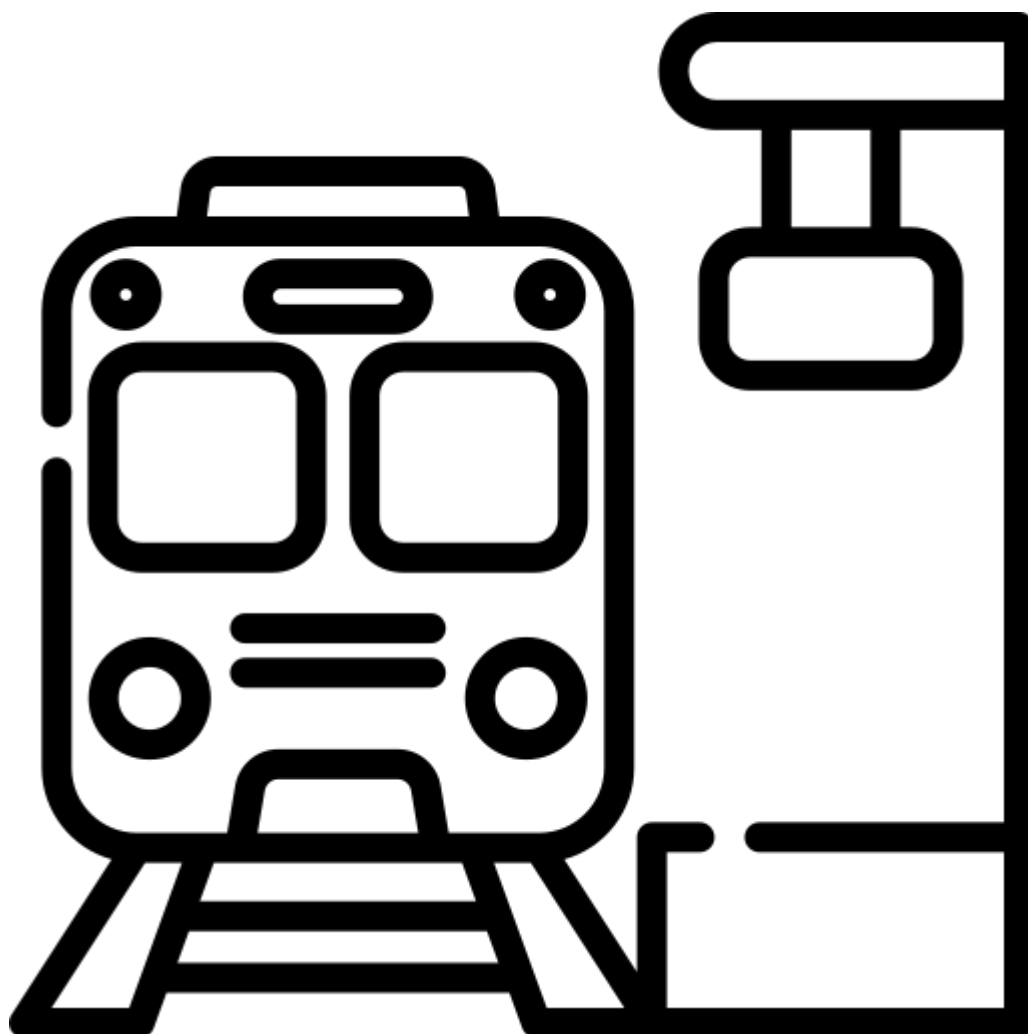


# TransportCompany



Authors: Jakub Szot, Dominik Piątek, Nadia Nowomiejska

## Table of Contents

Introduction .....	3
Overview .....	3
Objectives .....	3
Diagrams .....	4
Class diagram.....	4
Object diagram .....	4
Sequence diagram .....	5
Object-oriented programming mechanisms .....	6
Class definition.....	6
Encapsulation of data and methods .....	7
Inheritance .....	8
Aggregation .....	9
Polymorphism .....	10
User guide.....	11
Examples of use .....	12
First example .....	12
Second example .....	13
Third example .....	13
Github repository .....	14
Additional information .....	14

# Introduction

## Overview

**TransportCompany** is a Java-based agent simulation designed to simulate the behavior of a train network. The program uses various randomising mechanisms to set the number of passengers on each station. Stations were placed in a coordinate system, while the relationships between them were defined by a graph structure.

There are two types of stations - **CityStation** and **VillageStation** (highlighted in blue and red respectively in the picture below). The main difference between CityStations and VillageStations is that CityStations have a higher minimum of passengers and generate more passengers.

Additionally, there are two types of trains - **ExpressTrain** and **RegionalTrain**. ExpressTrains travel only between CityStations, while RegionalTrains can move between all stations.

The simulation iterates day by day and calculates the transportation company's budget at the end of each day. The result of the simulation is a .csv file summarizing the final budget of each day for which the simulation was performed.

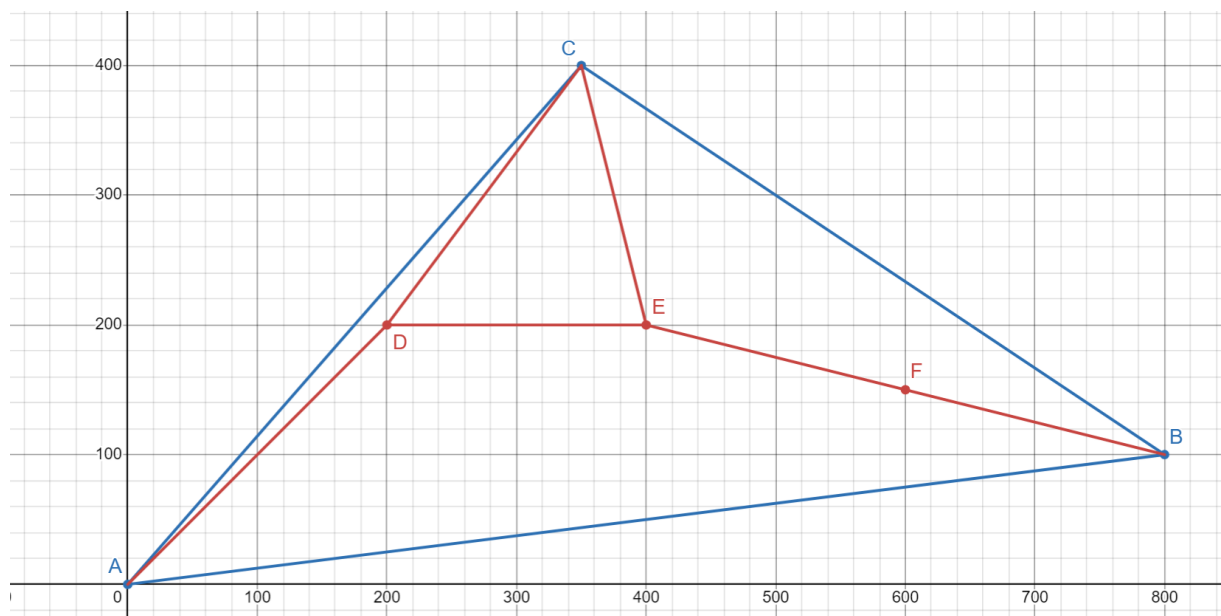


Figure 1 - Stations graph

## Objectives

The main objectives of this project are:

- Calculating the budget of the transport company with the given parameters.
- Accurately simulating a fragment of reality.
- Generating .csv files with the results of the simulation.

# Diagrams

## Class diagram

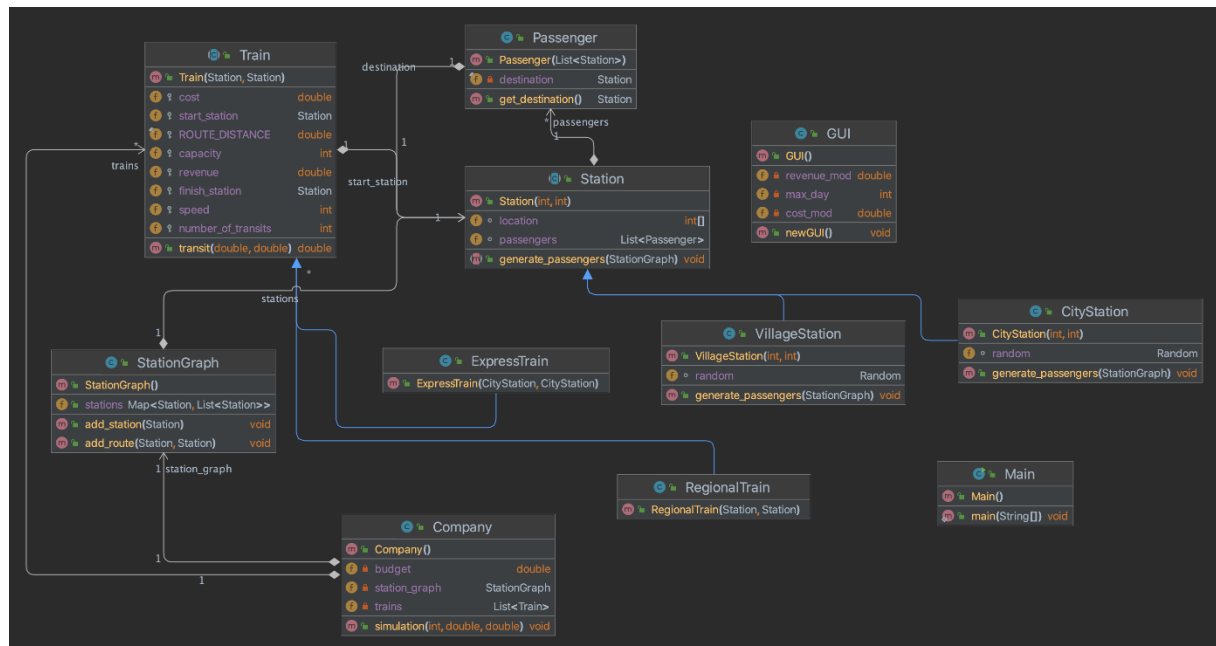


Figure 2 - Class diagram

## Object diagram

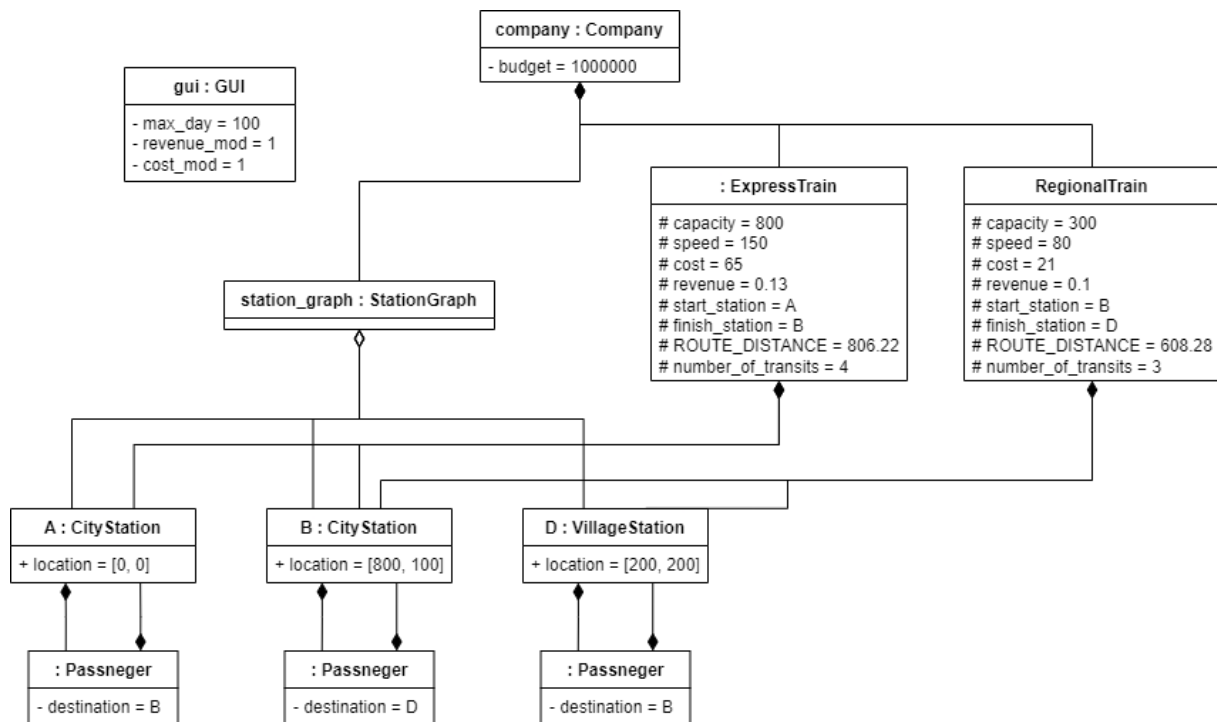


Figure 3 - Object diagram

## Sequence diagram

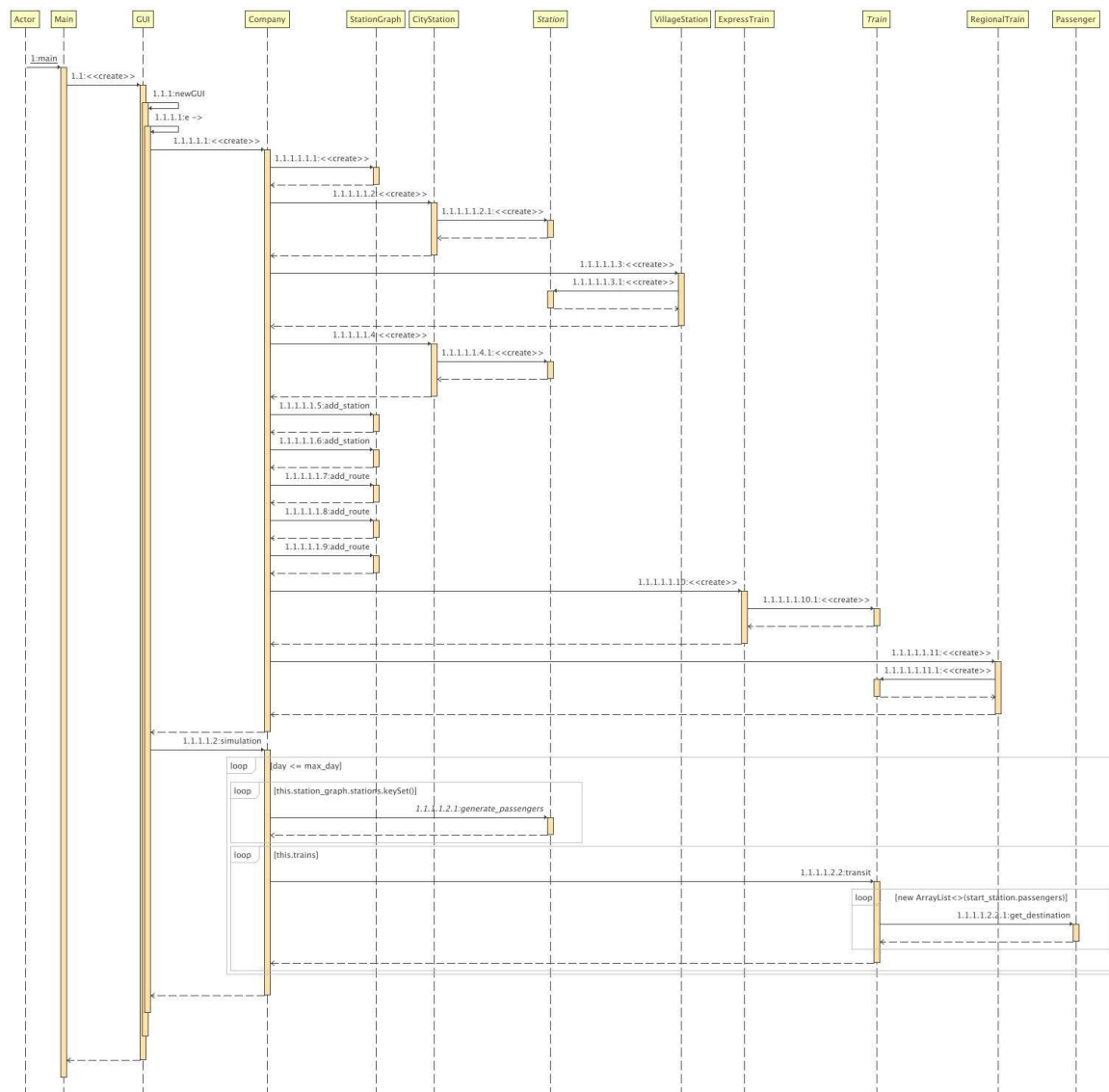


Figure 4 - Sequence diagram

# Object-oriented programming mechanisms

## Class definition

In object-oriented programming, a **class** is a template used to create objects. It defines the initial state through member variables and outlines behaviour through member functions or methods.

For instance, consider the **StationGraph** class. This class defines a template for creating objects that represent a graph structure of stations. It uses member variables to store station data and their connections, and provides methods to manipulate these data structures (**add\_station**, **add\_route**), and define behaviours related to station relationships within a transportation system.

Here's the **StationGraph** class example:

```
import java.util.*;

public class StationGraph
{
    public Map<Station, List<Station>> stations;

    public StationGraph()
    {
        stations = new HashMap<>();
    }

    public void add_station(Station station)
    {
        stations.put(station, new ArrayList<>());
    }

    public void add_route(Station from, Station to)
    {
        stations.get(from).add(to);
        stations.get(to).add(from);
    }
}
```

Figure 5 - "StationGraph" class

## Encapsulation of data and methods

In object-oriented programming, **encapsulation** is a fundamental principle that involves bundling the data and the methods that operate on the data into a single class. Encapsulation restricts direct access to some of the object's components, which can prevent accidental modification of data by an outside class. Instead, the variables are kept private, while the data is accessed and modified through public methods, providing a controlled and safe interaction with the object's state.

Consider the **Passenger** class. The data is kept private, ensuring it cannot be directly accessed or modified from outside the class. Instead, the class provides a public method, **get\_destination**, to access the destination.

```
public class Passenger
{
    private final Station destination;

    public Passenger(List<Station> available_stations)
    {
        Random rand = new Random();

        this.destination = available_stations.get(rand.nextInt(available_stations.size()));
    }

    public Station get_destination()
    {
        return this.destination;
    }
}
```

Figure 6 - "Passenger" class

## Inheritance

**Inheritance** in object-oriented programming is the mechanism of basing a class (subclass) upon another class (superclass), retaining similar implementation. The subclass acquires all of the properties and behaviours from its superclass, with a few exceptions.

Consider **Station** and **CityStation** classes. Inheritance in object-oriented programming allows the **CityStation** class to extend and inherit characteristics from the **Station** class. By inheriting from **Station**, **CityStation** gains access to its attributes and methods, such as location and passengers, while customizing its behavior through methods like **generate\_passengers**.

```
import java.util.ArrayList;
import java.util.List;

public abstract class Station
{
    int[] location = new int[2];

    List<Passenger> passengers;

    public Station(int x, int y)
    {
        this.location[0] = x;
        this.location[1] = y;
        this.passengers = new ArrayList<>();
    }

    public abstract void generate_passengers(StationGraph graph);
}
```

Figure 7 - "Station" class

```
import java.util.Random;

public class CityStation extends Station
{
    Random random = new Random();

    public CityStation(int x, int y)
    {
        super(x, y);
    }

    @Override
    public void generate_passengers(StationGraph graph)
    {
        int number_of_passengers = (int)((random.nextGaussian() + 1) * 1500);

        for (int i = 0; i < number_of_passengers; i++)
        {
            this.passengers.add(new Passenger(graph.stations.get(this)));
        }
    }
}
```

Figure 8 - "CityStation" class



## Aggregation

**Aggregation** is a type of relationship between classes in which one class has a reference to another class, but the second class can exist independently of the first.

For example: in the **Passenger** and **Station** classes, aggregation is illustrated by **Station** maintaining a list of **Passenger** objects (passengers). Each **Passenger** has a destination **Station** assigned randomly from a list. This setup allows **Passenger** objects to be associated with different **Station** instances while maintaining independence.

```
public class Passenger
{
    private final Station destination;

    public Passenger(List<Station> available_stations)
    {
        Random rand = new Random();

        this.destination = available_stations.get(rand.nextInt(available_stations.size()));
    }

    public Station get_destination()
    {
        return this.destination;
    }
}
```

Figure 9 - "Passenger" class

```
import java.util.ArrayList;
import java.util.List;

public abstract class Station
{
    int[] location = new int[2];

    List<Passenger> passengers;

    public Station(int x, int y)
    {
        this.location[0] = x;
        this.location[1] = y;
        this.passengers = new ArrayList<>();
    }

    public abstract void generate_passengers(StationGraph graph);
}
```

Figure 10 - "Station" class

## Polymorphism

**Polymorphism** in programming refers to the idea that objects of different types can be treated as if they're objects of a common class. This allows methods to work with different types of objects in a universal way because of their shared characteristics or interfaces.

Consider **Station**, **CityStation** and **VillageStation** classes. In the **Station** class, the **generate\_passengers** method was defined, which gained its unique use in the **CityStation** and **VillageStation** classes. In the **CityStation** class, a Gaussian distribution was used to draw passengers, while in the **VillageStation** class, a basic random function was used. This shows how both of the classes used a fundamentally similar method of the same name, but with different implementations, specific to their own characteristics.

```
public abstract class Station
{
    //...

    public abstract void generate_passengers(StationGraph graph);
}
```

Figure 11 - "Station" class

```
public class CityStation extends Station
{
    //...

    @Override
    public void generate_passengers(StationGraph graph)
    {
        int number_of_passengers = (int)((random.nextGaussian() + 1) * 1500);

        for (int i = 0; i < number_of_passengers; i++)
        {
            this.passengers.add(new Passenger(graph.stations.get(this)));
        }
    }
}
```

Figure 12 - "CityStation" class

```
public class VillageStation extends Station
{
    //...

    @Override
    public void generate_passengers(StationGraph graph)
    {
        int number_of_passengers = random.nextInt(600);

        for (int i = 0; i < number_of_passengers; i++)
        {
            this.passengers.add(new Passenger(graph.stations.get(this)));
        }
    }
}
```

Figure 13 - "VillageStation" class

# User guide

After starting the program, the following window is shown:

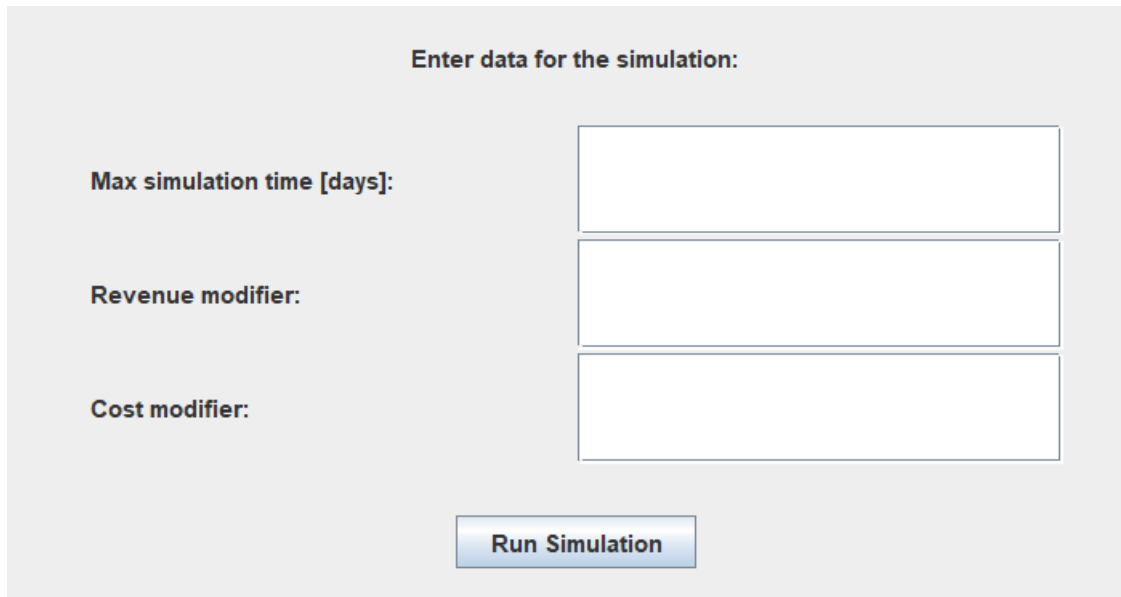
The screenshot shows a window titled "Enter data for the simulation:". It contains three labels on the left: "Max simulation time [days]:", "Revenue modifier:", and "Cost modifier:". To the right of each label is a corresponding empty text input field. At the bottom center of the window is a button labeled "Run Simulation".

Figure 14 - TransportCompany GUI

Enter the following values:

- **Max simulation time [days]** – maximum number of days for which the simulation will be performed
- **Revenue modifier** – revenue multiplier (per passenger per kilometer of travel)
- **Cost modifier** – cost per kilometer of travel

Click „Run Simulation“. After that, the following window will appear:

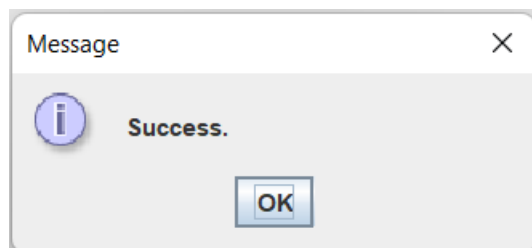


Figure 15 - TransportCompany GUI

In order to stop the program, close the window. The simulation results will be available in a .csv file in the folder where the program is located.

# Examples of use

Based on the resulting .csv files, charts were created to illustrate how the simulation works.

## First example

- **Max simulation time [days]:** 100
- **Revenue modifier:** 1
- **Cost modifier:** 1

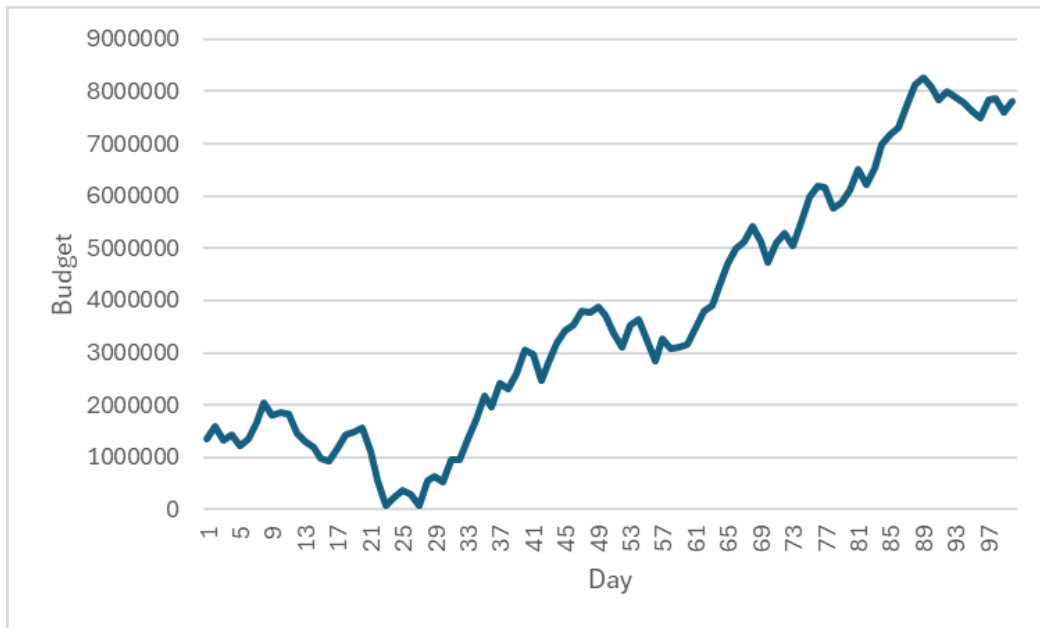


Figure 16 - First example chart

## Second example

- **Max simulation time [days]:** 100
- **Revenue modifier:** 1
- **Cost modifier:** 1.2

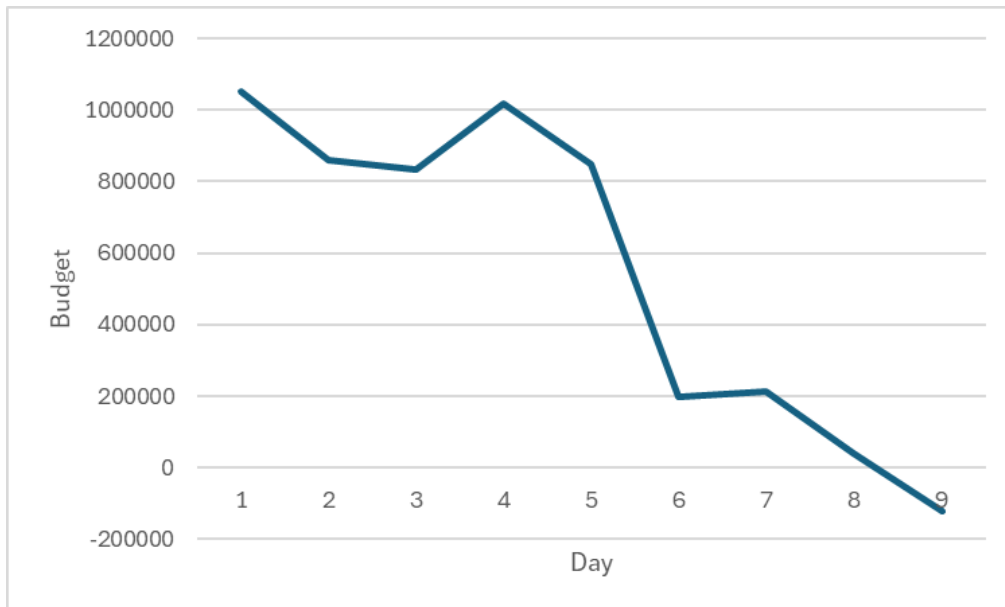


Figure 17 - Second example chart

## Third example

- **Max simulation time [days]:** 100
- **Revenue modifier:** 1.2
- **Cost modifier:** 1

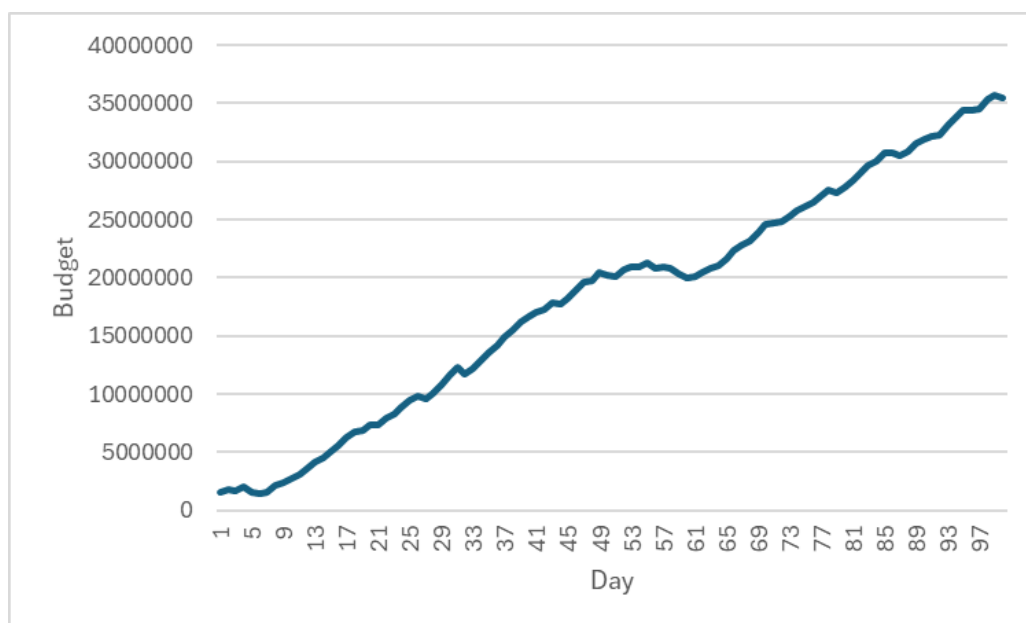


Figure 18 - Third example chart

## Github repository

Link to the github repository of this project:

<https://github.com/dekstr0metorfan/TransportCompany/>

## Additional information

Javadoc documentation is also included in the project folder.