**Trains**

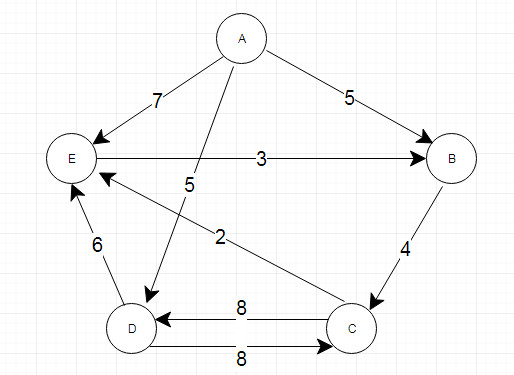
Analyzing the given Trains problem and based on these three pieces of information:

1. All of the tracks are 'one-way.'
2. A given route will never appear more than once.
3. For a given route, the starting and ending town will not be the same town.

I decided that the problem can be represented by a weighted directed graph with no loops -direct route between a town and itself-

Based on the given input: AB5, BC4, CD8, DC8, DE6, AD5, CE2, EB3, AE7

The graph can be represented as following:



For the OOP design for this problem I have:

2 main components to represent the entities:

1. **Graph**: represents the towns and the direct routes between each two towns.
2. **Town**: represents a town and its neighbors.

5 components to solve the 10 questions:

1. **RouteDistanceCalculator**: solves the first question 1, 2, 3, 4, 5
2. **TripsCalculator**: solves questions 6, 7.
3. **ShortRouteCalculator**: solves questions 8, 9.
4. **RoutesCalculator**: solves question number 10.
5. **HelperCalculator**: provides helper methods to the calculators.

1 component to read the input file:

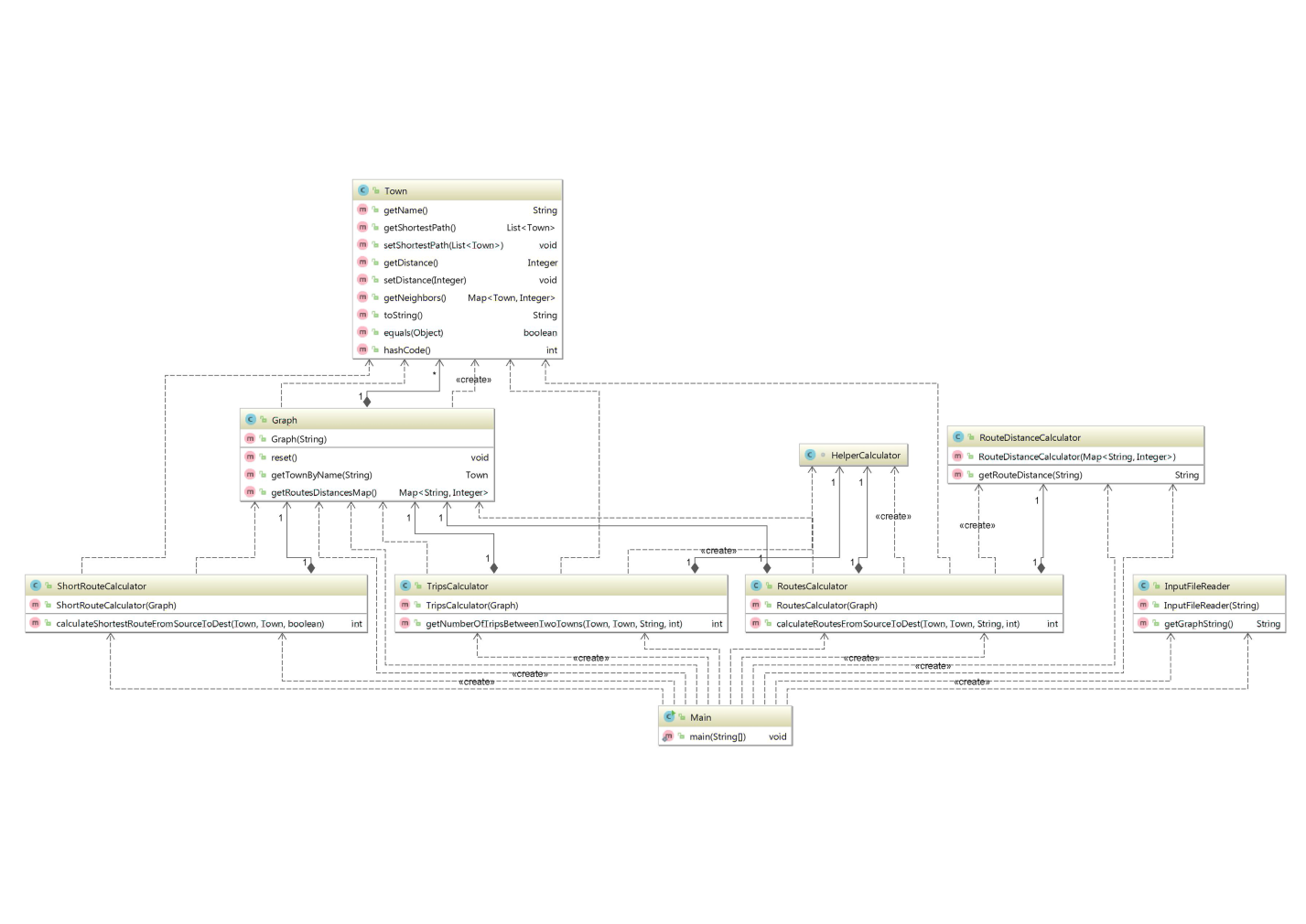
- **InputFileReader**: reads the input file and provides the input String that represents the graph.

1 component to run the application:

- **Main**: the main class with the main method that reads the input data, initializes the application and calls the calculators to solve the 10 questions.

I represented the graph using an adjacency list to store the towns, their neighbors and the distance between each town and its neighbors.

Also, I stored the routes & their distance in a map to be used later will solving questions. i.e: to check if a route exists, to get the distance of an existing route.



**The class model of the application -also attached to the project docs-**

**The ideas of the 10 questions are:**

**Q1, Q2, Q3, Q4, Q5:**

* Initialize a total sum of the route to be 0.
* Divide the given route to sub-routes.
* Check if the sub-route exists.
* If yes, add the length of it to the total sum of the full route and continue to the next sub-route.
* If not, return -1.
* If the return is > 0, return the string representation of the value. If not, return “NO SUCH ROUTE”.

**Q6, Q7:**

* Traversing the graph starting from a given source town.
* Counts the stop at each town till the given condition to stop is met.
* Filters the numbers of stops at the given destination based on the given condition and returns the number of trips that matches the condition of number times of stops.

**Q8, Q9:**

* Applying Dijkstra algorithm to traverse a Graph starting from a given source town and calculates the length of the shortest path distance of all towns then returns the length of the shortest path from the source town to the destination town.
* If the source town is the same as the destination town, I perform Dijkstra algorithm starting from the source town first and I cache the distances then I start to perform Dijkstra from each neighbor and sum the shortest route length to the destination with the cached distance of the neighbor, by comparison, I can get the shortest route length from a town to itself.
* i.e: Start from “B” to “B”
* “B” has neighbors “C”, “D”
* Short distance from “B” to “C” = 3, short distance from “B” to “D” = 5
* Start from “C”, short distance to “B” = 6
* Start from “D”, short distance to “B” = 1
* Distance from “B” to “C” to “B” = 3 + 6 = 9
* Distance from “B” to “D” to “B” = 5 + 1 = 7
* Then shortest distance from “B” to “B” = 7

**Q10:**

* Applying Depth First Traversal algorithm to find all routes from a source node

in a graph to a destination node.

* If the source town is the same as the destination town, I build routs from the neighbors of the source town to it and I add the source town to the start of each route.
* To apply the given distance condition, I consider unique routes & unique combination between them to calculate all the possible routes based on the distance condition.