# Objectville RouteFinder

## Needs List

1. Store Objectville's complete network of subway lines and all of the stations.
2. Subways can go backwards and forwards.
3. Determine a route of travel given a starting station and ending station.
4. Present summary describing:
   1. Which lines to take
   2. Which stations are passed
   3. Which stations must be used to transfer

## Feature List

1. We have to be able to represent a subway line and stations along the line.
2. We must be able to load multiple subway lines into the program, including overlapping lines (they share the same station).
3. We need to be able to figure out a valid path between any two stations on any lines.
4. We need to be able to print out a route between two stations as a set of directions.

## Use Case Diagram

## Modules

1. **Subway**
   1. Has all the code that represents stations, connections between stations, and the entire system itself.
   2. It also knows how to get directions.
2. **Loader**
   1. Could be loaded from a file, user input, database, etc.
3. **Printer**
   1. Handles printing the subway to any device or format needed.
4. **Tester**
   1. Contains all tests of the system.
   2. Confirms that the program works without bugs.
   3. Demonstrates to the customer the program functionality.

## Understanding the System

* **What is a station?**
  + A point on the map with a name.
* **What is a connection between two stations?**
  + A path that joins two stations.
* **What is a line?**
  + A series of connections.

## Use Case: Load Network of Subway Lines

1. The administrator supplies a file of stations and lines.
2. The system reads in the name of a station.
3. The system validates that the station doesn't already exist.
4. The system adds the new station to the subway.
5. The system repeats steps 2-4 until all stations are added. This is indicated by the encounter of the first space, or the name "Line" in the string.
6. The system reads in the name of a line to add. The system validates that the line doesn't already exist.
7. The system reads in two stations that are connected.
8. The system validates that the stations exist. The system also checks that the two stations are different stations.
   1. Additional check can be done to ensure that each station added is unique to the list, except for the first entry. This enforces that a line doesn't double back on itself.
   2. The list is considered closed when the first item is next encountered.
9. The system creates a new connection between the two stations, going in both directions, on the current line.
10. The system repeats steps 7-9 until the line is complete.
11. The system repeats steps 6-10 until all lines are entered.

## Textual Analysis

### Nouns

* ~~Administrator~~
* ~~File~~
* Station(s)
  + Name
* Line(s)
  + Name
* ~~System~~
* Subway
* Connection

### Verbs

* Supplies (a file)
* ~~Reads in (the station name)~~
* Validates (station unique)
* Adds (new station)
* ~~Repeats (steps 2-4)~~
* ~~Reads in (the line name)~~
* ~~Reads in (two station names)~~
* Validates (stations exist)
* Creates (a new connection between the two stations)
* ~~Repeats (steps 7-9)~~
* ~~Repeats (steps 6-10)~~

A line class will not be created because due to the behavior of the system and the customer needs, we can get all of the information we need from the subway composed of stations and connections, if the connection names have the name of the associated line. A line class in this case is redundant.

## UML Diagram

## Use Case: Get Directions

1. Travel agent chooses a starting station and destination station from the system.
2. The system validates that both stations exist.
3. The system calculates a route from the starting station to the ending station.
4. The system prints out the route it calculated.

## Textual Analysis

### Nouns

1. Travel agent
2. Starting station
3. Destination station
4. System
5. Route

### Verbs

1. Chooses (stations)
2. Validates (stations)
3. Calculates (a route)
4. Prints (a route)

## UML Diagram