1)

Rationale for representation of the MultiGraph:

As a group we decided that an ArrayList of Nodes and ArrayList of Edges would be the easiest, but not the most efficient way of implementing our MultiGraph. We could’ve used a HashMap representation if we had more time, but an ArrayList serves our purpose fine.

All Nodes and Edges are represented globally, and only actually modified in the addEdge and addNode methods.

We thought it would be best to have Nodes as an object with an ID and a Name, and Edges with a String label and two Nodes. This saved us time as well as helping abstract the concept of the MultiGraph further.

Our Multigraph implements MultiGraphADT. MultiGraphADT also has findRoute and getPath. Although this is somewhat against what a MultiGraph does conceptually, it reduced semantic coupling between the BostonMetro class, and MultiGraph.

Changes made to design based on feedback:

We changed Edge to use actual Nodes instead of ints for its representation of nodes.

Parser’s createMap method is now a void which takes in a MultiGraph object and has it filled with nodes and edges. This reduced coupling between the Parser and the BostonMetro class.

We removed the nNodes and nEdges methods as they were not used. The return type of addEdge and addNode were changed to void as their previous boolean return type were not used. We also reduced the number of global variables in the MultiGraph and BostonMetro classes.

2)

1. Interfaces:

MultiGraphADT- This interface contains method declarations that allow for the handling of data in the context of a MultiGraph.

Edge-The Edge interface contains method declarations for retrieving data for an edge between two Nodes.

Node-The Node interface represents a typical vertex in a graph, it only has two methods for retrieving its name and unique identifier.

Classes:

MultiGraph- Implements MultiGraphADT and provides implementation of each function from the interface. It creates a collection of edges and nodes, additional methods are also provided to aid the functionality for the searching and creation of routes. This is used by BostonMetro.

BostonMetro- Creates a new MultiGraph using the Parser createMap (see below) method, this class also contains all the console interaction for a user to find a route between stations. BostonMetro gives two Stations to MultiGraph to find a route between. BostonMetro then gets returned a list of directions as a String, and prints them to the user.

Line- This class implements Edge, taking in the name of the line, the source vertex and the destination vertex. BostonMetro sends Lines to the MultiGraph it uses, as a Line implements Edge, and, as such, is an accepted parameter for its methods (e.g addEdge). Line has an overridden hashCode and equals method.

Station-Implements Node. A station has the exact same information “blueprint” as a Node. Station also has an overridden equals and hashCode method, use for future changes involving hashmaps. BostonMetro also sends Stations to the MultiGraph it uses, as a Station implements Node, and, as such, is also an accepted parameter for its methods (e.g addNode). Node has an Overridden hashCode and Equals method for usage in HashMaps.

Parser- Parses information from a given file about stations and lines. BostonMetro uses this to fill a MultiGraph which is used later on for when a user wants to find routes. The classes main createMap method also checks for errors in the grammar of the file.

b) One relationship is that BostonMetro uses a MultiGraph which implements MultiGraphADT, as the BostonMetro and MultiGraph have the same concept of nodes and edges (in the context of the BostonMetro; stations and lines).

Line also implements Edge, as they contain the same components but Line in the context of subway lines. This is the same case for Station implementing Node, the stations of the Subway being like the vertices of a graph. Line and Station have some extra overridden hashCode and equals methods, for testing and future implementation.

c)

MultiGraph:

addEdge- Adds the given edge to the list of edges.

addNode- Adds the given node to the list of nodes

isEdge- Verifies whether or not two nodes exist in an edge in the list

getNeighbours- Returns a list of nodes that are neighbours to a supplied node.

getNodeByName- Returns a node with a matching name.

getNodeById- Returns a node with that ID. Used for when we need to get a St.PaulStreet

BostonMetro:

run()- creates a new MultiGraph

Line: (implements Edge)

getLabel- Returns the colour of the line

getSrcNode- Returns the station that a train is coming from.

getDestNode- returns the station that the train will be going to next.

hashCode- Used so that a HashMap (or any method which uses hashing) correctly knows how to compare two values to another(yet to be fully implemented)

equals- Compares 2 nodes to check if they are equal.

toString- For testing purposes, self explanatory.

Station: (implements Node)

getName – Returns the stations name/label.

getId – Returns the stations ID.

hashCode- Used so that a HashMap correctly knows how to map values to another(yet to be fully implemented).

Parser:

createMap- Reads a file of lines and stations to fill a MultiGraph which is passed as a parameter.