Group w01 –

Kuen Wai Chan, Ben Hubbard, Fraser Steel, Elena Xiao

**Overview**

In order to promote decoupling within the project, and potential future re-usage of certain classes (such as those for the Multigraph), the project has been separated into two halves. The two halves consist of those representing functionality of the Metro System, and another outlining the Multigraph. Furthermore, to attempt to structure the project a bit better we decided to aim for a Model-View-Controller approach.

* Model being the multigraph and the data that is held in the nodes and edges.
* View being the ConsoleIO which was responsible for reading in user input but also for big jobs like printing out the route in an easy to read way. As this was just a console user interface and no Gui there are a few system prints in the controller class as we didn’t feel it was entirely necessary to ask the ConsoleIO to print every line to the user.
* Controller being the Controller class which would take the input read in by ConsoleIO and handle it in relation to the data held in the graph. For example, in order to verify the user has given a valid station name it made sense to us to have somewhere would be able to query the data and return a response.

We decided to implement Stations and Lines metro side instead of Nodes and Edges graph side as we felt that in a larger program it would give more flexibility in what data could be stored on the nodes implementation as you wouldn’t be able add more fields to a Node class as it wouldn’t necessarily make sense to the multigraph.

**UML Description**

**Role of Interfaces**

IMultigraph: Outlines the functionality of a multigraph. Public methods will allow for the addition of Nodes and Edges to the multigraph. Multigraphs will also be capable of returning a List of Edges representing a path between two Nodes. The List being an implementation of the interface java.util.List.

IEdge: Outlines the functionality of Edges, used within multigraphs. Methods may include returning the label of the edge, or returning one of the nodes that the edge connects to (two separate methods are required, one for each node).

INode: Outlines the functionality of Nodes, used within multigraphs. This may include returning the values of the ID.

**Role of Classes**

MultiGraph: A concrete implementation of the IMultigraph interface. This would allow the program to instantiate an instance of Multigraph and use it in place of IMultigraph references.

ParentNodeRecord: Nested class of MultiGraph, is used as a simple data structure to store information when constructing paths between INodes. Information includes the parent INode and the IEdge connecting the INode to the parent.

Line: Likewise, this provides an implementation of the IEdge interface.

Station: This provides an implementation of the INode interface. Stores the name.

MetroApp: This class must be run by the user to initialise the program. It contains the only main method within the project.

MetroMapParser: This class (which is provided) will parse the given text file in order to fill a MultiGraph with information regarding the Boston Metro system (such as the stations and track lines).

ConsoleIO: This class acts as the View within the Model-View-Controller; it will manage user messages and retrieve user-input through the console.

Controller: This class acts as the Controller within the Model-View-Controller; it will parse the user input from an instance of User Prompt, and manage the Metro system app in accordance to the input. Controller now holds Multigraph rather than the initial design of using Metro.

**Relationships between interfaces and classes**

Interfaces: IMultigraph, IEdge, INode

Implementations of Interfaces: MultiGraph, Line, Station (respectively)

Within the Multigraph half of the program: Multigraph (the implementation of IMultigraph) will store a list of IEdges and a HashMap of ids (Key) and INodes (value),

IEdges will store the two INodes it is connected to.

Within the Metro half:

MetroApp creates an instance of MetroMapParser and Controller.

Controller will hold some implementation of IMultigraph rather than the initial design of Metro holding it. We removed the metro class.

Controller also has an instance of ConsoleIO, both of which just refer to Lines and Stations.

MultiGraph will initialise ParentNodeRecords while executing the getRoute() method. These ParentNodeRecords contain information of an INode and its parent within the path that getRoute() will return. A ParentNodeRecord is mapped using a HashMap to a Node.

**Method descriptions**

IMultigraph/MultiGraph:

* addNode(INode) – Allows for the multigraph to be populated with nodes.
* addEdge(IEdge) – Allows for the multigraph to be populated with edges.
* getRoute(INode, INode) – Given two nodes, it will return a List (java.util.List) which

contains a series of edges in a path between the two nodes.

* getNodes() – returns a copied list with nodes
* getNode(int) – Returns a node with a given ID.
* Successors(Node) – Returns a list of edges connected to a given node.

MultiGraph:

* Private checkEdgeExists(IEdge) – returns true if edge exits with the same label between the same two nodes. Returns false if not.

ParentNodeRecord:

* getParent() – returns the parent INode
* getEdge() – returns the IEdge which is correlates to the Parent INode.

IEdge/Line:

* getNode1() – returns the INode of one of the nodes the edge connects to.
* getNode2() – returns the INode of the second node that the edge connects to.
* getlabel() – returns the label of the edge.
* getOtherNode(int) – returns the node which doesn’t have the id passed in.

INode/Station:

* getID() – returns the ID of the node
* setName(String) – sets the name of a station
* getName() – sets the name of a station

ConsoleIO:

* prompt(String) – prompts the user with a given message past in as a parameter then returns the user input.
* printRoute(List<Line>, int, int) – displays the start and end points and uses formatRouteList for the middle (used to reduce length of the method)
* formatRouteList(List<Line>)- formats the main route in a readably way.
* printNumberOfStops(String, int) – prints the number of stops between a section of the journey. (used to reduce duplication)
* printStationDetails(Line,int) – prints the information about the station with the given id on the given line.
* PrintCollection(Collection<?>) – Used to print out a collection of all elements. (this was initially specific to Strings but felt it wasn’t necessary to restrict it, so now just would use the toString() method on other objects)

Controller:

* Run() – where the main loop for the reading user input is based, it was placed in here as it includes some validation(of whether it was a valid station) and wanted to make sure the ConsoleIO didn’t know about metro or the controller.
* validateInputStation(String)- validates the input from the user and provides some error checking. If the user doesn’t provide the station with a capital letter at the start it will automatically capitalise it.
* manageStationNotValid(String) – If the station it not valid an error message will be printed so the user is made aware. A list of suggestions will also be collected based on the input being the prefix of any station names.
* manageDuplicateStation(List<Node>) – Used to manage when a duplicated station is detected. A choice of what station they meant will be displayed.
* getStationsWithName(String) – this will query the list of nodes returned from multigraph and add each node to a new list depending on if it has the same name as the input.
* getStationNamesWithPrefix(String) – this will do similar as above but instead on filtering based on if the name is equal, its if the name of the node starts with the input string from the user.

MetroMapParser:

* generateGraphFromFile() – this returns some concrete implementation of IMultigraph filled with Stations and Lines based on some txt file that was already loaded.
* createStationIfNotPresent(int, IMultigraph) – Creates a station if it is not already present in the graph.

**Changes from initial design**

* We removed the Metro class from our initial design and allowed controller to hold the multigraph as the class wasn’t really required and had just ended up having the same methods as the graph and was just in the way.
* Moved methods from Station to INode as we felt that it made sense that every node would have a name.
* Instead of having Controller have an instance of MetroMapParser we moved that up to MetroApp as this gave Controller less to know about and also because we had gotten rid of Metro class from before.
* Added several methods to different classes in order to split up the work.
* Provided more error checking so that the users input would work even if the put spaces between input and didn’t capitalize their entry.
* Improve the look of the console output so it is easier for the user to use, and also easier for them to understand the journey plan.
* Added Nested class into Multigraph used for storing information about paths constructed by getRoute().