MBTA Software

- Requirement Specification & Design Document -

TEAM:
Peter Wilson
Jeremiah Froh
Cosimo Leone

DATE: 12.3.12
VER: 1.2

Requirements Specification

Software Overview:

The MBTA trip planner allows users to see live train locations, train arrival times, and schedule routes. This will be accomplished utilizing the JSON live T data feed the MBTA system currently exports on their website. The software will harvest all the data available on Boston's MBTA system and deliver a 'smarter' experience for the T rider, deriving real-time trip routes based on user input and train GPS data.

User features have been prioritized by functionality that is essential, desirable, and optional. They will be highlighted in this overview along with a detailed report on achieving them in hardware.

High-Level Requirements:

User Stories:

- The user wants to know where she can go using the T. (Essential)
- The user wants to know the current location of all trains. (Essential)
- The user wants to know when the next trains get to stop A. (Essential)
- The user wants to know her options for getting from stop A to stop B. (Essential)
- The user wants to know her options for getting to an ordered list of stops. (Desirable)
- The user wants to know her options for getting to an unordered list of stops with specified starting and/or ending points. (Optional)
- For any trip on the T, the user wants the option to specify departure and/or arrival times.(Desirable)
- For any trip on the T, the user wants to know the fastest route, the earliest departure, the earliest arrival, and fewest transfers. (Desirable)

- The client wants to test the system with old data. (Essential)

Constraints
- The system should be based on live data from the MBTA (http://www.mbta.com/uploadedfiles/Description.pdf).
- The only information the system should access online is MBTA data.
- The system should be written in Java and run on CCIS Linux machines with only their standard software and JSON library in the course directory.
- The system should accept test data from file input.

User Interface
- The system should include a user-friendly GUI. (Optional)

Use Cases:

01: User wants to know where she can go using the T.
Actor: User who wants to see train destinations.
User: Launches application.
System: Displays main window titled Plan a Trip which includes a static map and sidebar along with three drop boxes.

User: Selects dropbox (Red, Blue, Green) for destinations by line.
System: Produces selectable list of all available stations based on line chosen.
User: Repeats above step for all destinations.

02: The user wants to know the current location of all trains.
Actor: User who wants to see train locations.
User: Launches application.
System: Displays main window with live location of all trains

03: The user wants to know when the next trains get to stop A.
User: Launches application.
System: Displays main window
User: Types desired station into text form; clicks add.
System: Click handler adds form value to the trip planner queue.
User: Clicks on the ‘plan trip’ button to display arrival times of next train.
System: Returns secondsAway from desired station.

04: The user wants to know her options for getting from stop A to stop B.
User: Launches application.

System: Displays main window
User: Selects start station from one of the dropboxes.
System: Click handler adds form value to the trip planner queue.
User: Selects end station from one of the dropboxes.
System: Click handler adds form value to the trip planner queue.
User: Clicks on the ‘plan trip’ button
System: Returns arrival times and route

05: The client wants to test the system with old data.
Currently no switch and has to be handled manually in code

User: Launches application - with test file loaded
System: loads main display and populates with old data
User: - normal operation -

06: The user wants to know her options for getting to an ordered list of stops
User: Launches application.
System: Displays main screen
User: Check ordered selection check box
User: Selects start station from one of the dropboxes.
System: Click handler adds form value to the trip planner queue.
User: Selects end station from one of the dropboxes.
System: Click handler adds form value to the trip planner queue.
User: Clicks on the ‘plan trip’ button
System: Returns arrival times and route

Detailed Requirements:

The user wants to know where she can go using the T.
This will be displayed through drop boxes populated by station lists.

The user wants to know the current location of all trains.
Train locations will be displayed on the live map. They are also available in list format pulled from the secondsAway field in the JSON data.

The user wants to know when the next trains get to stop A.
Once user has input stop A, Display next available train based on secondsAway

The user wants to know her options for getting from stop A to stop B.

This will require the first calculation algorithm. For this our data-structure will be implemented and a DFS search to calculate (A)->(B)

The user wants to know her options for getting to an ordered list of stops.
An ordered list of stops (A)->(B)->(C) can be viewed as (A)->(B); (B)->(C)
and will use the same algorithm for computation

The user wants to know her options for getting to an unordered list of stops with specified starting and/or ending points.
Unordered stops requires analyzing multiple options. This is more complex than simply traversing (A)->(B)->(C)

Start(A)->(B)->(C)
Start(A)->(C)->(B)

For any trip on the T, the user wants the option to specify departure and/or arrival times.
Time will be the driving factor on this calculation and will have to use the specified time rather than the secondsAway variable. This will also require a way to input time parameters.

For any trip on the T, the user wants to know the fastest route, the earliest departure, the earliest arrival, and fewest transfers.
Since we are utilizing a greedy algorithm for our time weight calculations, our app will default to the fastest route, with the earliest depart and earliest arrivals. Fewest transfers will be slightly more complex to implement

The client wants to test the system with old data.
This is easily achievable as we will be testing our own algorithms with the test data first.

CONSTRAINTS:
The only information the system should access online is MBTA data.
An error is thrown when our input files are not what is expected. Fields not properly formatted or the input file is corrupt will also throw errors.

The system should be based on live data from the MBTA
System defaults to live settings and throws above error when necessary.

The system should be written in Java and run on CCIS Linux machines with only their standard software and JSON library in the course directory.

All milestone code and documentation has and will be tested in the CCIS labs

The system should accept test data from file input.
A parser has been included for CSV files to handle file input.
JSON files are converted to CSV as well then handled by the same parser.

User Interface: Graphic User Interface (Java Swing)
Is not only visually appealing but intuitive as well. The Java listener waits for the the form prompt and drop down selection boxes for input. These buttons and input methods are common online for user interactivity and will feel comfortable to the any user familiar with web interfaces.

The GUI is coded independently of the back end Algorithms to ensure modularity. Optimizations in the Algorithm can be implemented without corrupting the User Side and vice versa. This allows for scalability of the software in the future with less overhead.

System: CCIS Linux Machines (default)
Language: Java with JSON library
Internet: Required for JSON feed from MBTA host connection
Performance: Timely calculations desired

Design Document - Software Architecture

Inputs:

GUI Form Values

The user has a finite set of lines and trains to choose from and is prompted with a GUI that has three drop boxes. Each containing all the stations available on that line. A listener will wait for the next action. Red line Drop Box

```
JComboBox redDropdown = new JComboBox(RED_STATIONS);
               redDropdown.setForeground(MYRED);
               redDropdown.setMaximumRowCount(15);
               redDropdown.addActionListener(comboBoxListener);
```

The user can add trains to the trip by either clicking on the name in the drop box, or by typing the name and clicking add. The value of the selection is passed to the factory for input. The actual call to action, which is the plan trip button, will start the process and produce the output desired.

GUI Form DATA -> CONTROLLER -> ALGORITHM -> GUI

MainWindowGUI behaves like main and is the central call for all functions.
HandScrollListener waits for input through either add button or mouse click.
Constants are loaded here for the GUI to use. (e.g. stations)

MBTA Data

The feed data will be parsed and objectified into our java program and is never actually seen by the user.

Feed URL:

```
URL redData = new URL("http://developer.mbta.com/Data/Red.txt");
URL orangeData = new URL("http://developer.mbta.com/Data/Orange.txt");
URL blueData = new URL("http://developer.mbta.com/Data/Blue.txt");
```

If there is a file error the program currently just prints the stack

```
catch(IOException ex) {ex.printStackTrace();}
```

Static File:

```
URL blueData = new URL("local/test.txt");
```

For testing, provides consistent results and proper formatting

User Interface:

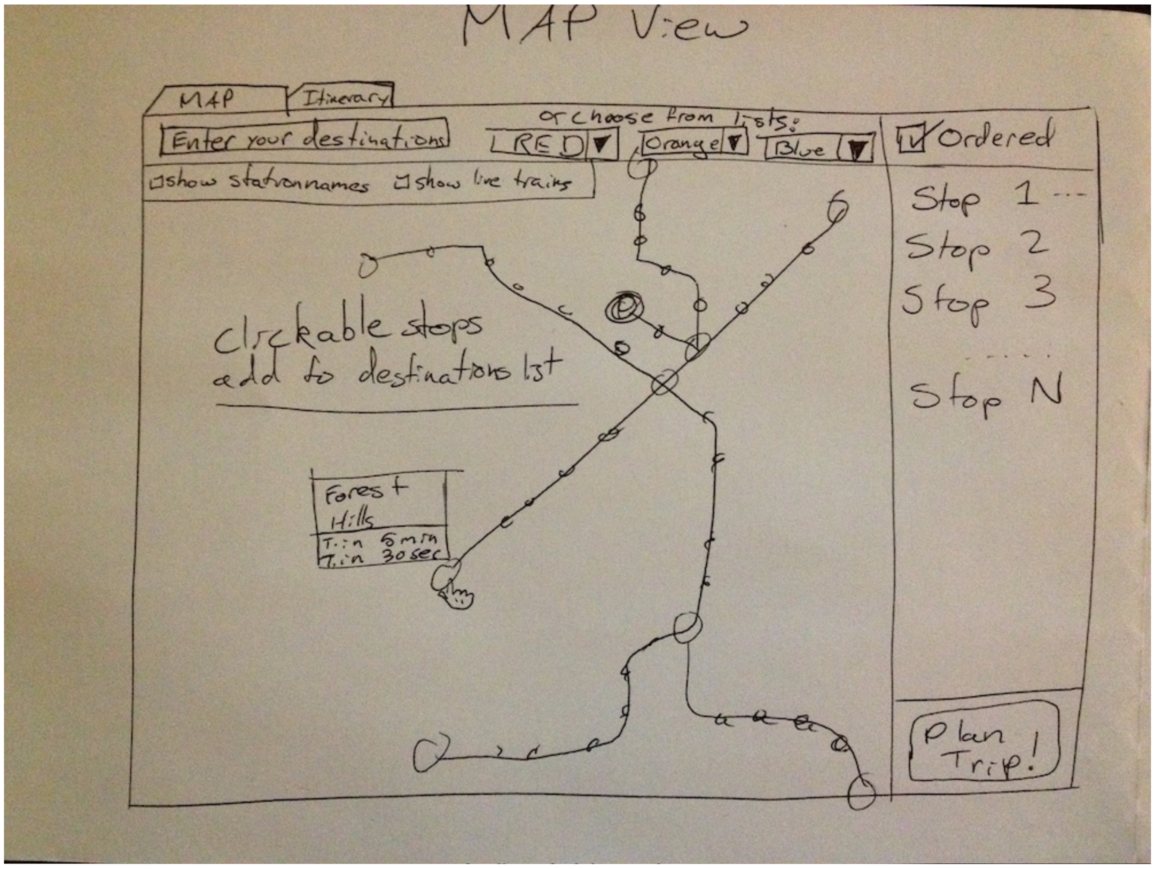
Using Java Swing

Modular and should be independent of Algorithm base code.

There are 3 tabs at the top of the app to control User View.

MAP VIEW:
- Graph of Stations
- Text Box for station manual add feature
- Drop boxes for each line with Add button for more stops.
- Selection box for (sort, unsorted) options

[text_form] [ADD]   [RED_LINE] [ORANGE_LINE] [BLUE_LINE]   [[]ORDERED]

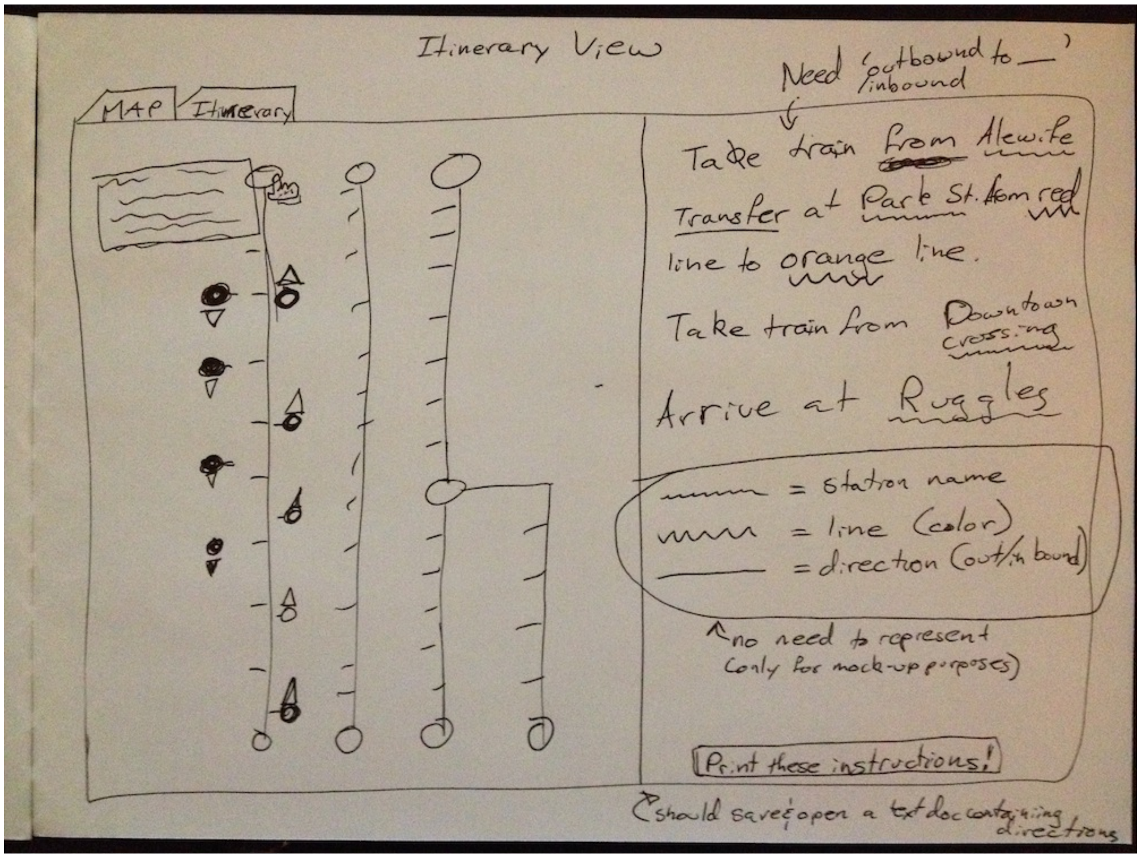


[PLAN TRIP]

The screen is split with a 80/20 ratio of map to list destinations.
80% map screen has scrolling options enabled.

ITINERARY VIEW:

Displays the route step by step.


Itinerary View
MAP
Itinerary
Need 'outbound to ___' /inbound
Take train from Alewife
Transfer at Park St. from red line to orange line.
Take train from Downtown crossing
Arrive at Ruggles
= station name
= line (color)
= direction (out/in bound)
no need to represent (only for mock-up purposes)
Print these instructions!
should save & open a txt doc containing directions

Calculations start at the press of 'plan trip' based on the number of stops and options selected. Results will be processed and output to the GUI on this itinerary screen.

Outputs:

As stations are selected they will appear on the side bar as output but no calculation's have actually taken place yet. They're actually input. Trains are displayed on live map with real-time locations every X seconds. This is updated automatically and will not require any user input .

The output derived from computation is first done in the backend then mapped to the GUI to be viewed by the user.

Once trip is processed depending on options.
Stations will be listed in Google maps fashion.

1. START (Station A) 00:00

2. END   (Station B) 00:01

Any transfers or stops will be listed sequentially with simple directions to next direction.

1. START (Station A) 00:00
2. Stop   (Station B) 00:01
3. END   (Station C) 00:02

Data Structures:

Graph of Nodes (stations) with ArrayList of stations and time

```
public class Graph {

    private ArrayList<Station> stations = new ArrayList<Station>();
    private ArrayList<Edge> edges = new ArrayList<Edge>();
```

Edges are defined as routes the Train can take from the current node.

```
public class Edge {
       private int weight;
       private String line;
       private String tripID;
       private Station startStation;
       private Station endStation;
```

Filled from JSON, from which we can add the weights to the graph. Each station is treated as a node. The weights are derived from current seconds away from MBTA feed data field and applied to each 'edge' in the graph. In this case either inbound or outbound from the given station.

Connection stations will have multiple in/outbound routes and end of the line stations will have only one in and one out.

Whichever permutation produces the lowest weight is the fastest time route*
*Greedy Algorithm

Algorithms:
DFS (Depth-First Search)

```
public Pathway<TrainConnection> depthFirstSearch(Station startStation, Station endStation, int
departByTime, int arriveByTime) {

Pathway<TrainConnection> pathway = new Pathway<TrainConnection>();
```

USED to create new ‘Pathway
’

To Reduce the DFS time:
Possibly initialize the 8 possible graphs
and when asked to produce a plan only search the relevant graph
For ordered list:
(A) -> (B)

first train arriving at A will be your depart time and arrival at B will be static.

Greedy Algorithm:
The greed factor drives the optimal route by always choosing the closest train. The T path is essentially linear, so the it chooses the lowest weight based on secondsAway from that station. This works because the user will most likely not want to wait longer than necessary for a train when scheduling isn’t imperative.*

*Sometimes this still equals long wait times at some stations that may not be feasible for a rider but will result in getting to destinations eventually.

(A) -> (B) ->(C)
We use the same algorithm to compute multiple stops. It simply recursively calls route(A B)

route(A B)
route(B C) ;; uses last stop as new start

For unordered list:
- Start or End is Stated - Then lock those down.
- Otherwise, do the DFS for all permutations of the list.
- does not account for time away of trains

Unordered List with Time sensitivity:
- Use specified times instead of arrival times based on trains secondsAway

MAKEFILE
Compiles the Program and packages all necessary Class Path’s to run correctly.
$ make test - to run the test suite
$ make run - to run application