Chris McConnell
Nikita Filatov
Ed Nunez
Adam Gressen

MBTA Trip Planner Software Architecture
Version 1.1

1. **Algorithms and Live Data Access**

The MBTA Trip Planner needs to quickly sort through MBTA data and use an algorithm that will find routes based on all of the options outlined in the Software Requirements Specification. The Trip Planner will follow the Model-View-Controller (MVC) software architecture. This architecture allows for separation between the different layers of the software. By separating the application into logic and view layers the code is more flexible and can be expanded more easily if anything needs to be added later. With the MVC architecture in place we can place all visual components in the View class, our data in their respective classes, and handle route planning algorithms in the TrainGraph class.
The following is a list of inputs and whether that input is required or optional in generating a trip:

1. MBTA data from website or test data file (required)
2. Starting location andEnding location (required)
3. Stops between starting and ending locations (optional)
4. Fastest route Boolean (optional)
5. Earliest departure Boolean (optional)
6. Earliest arrival Boolean (optional)
7. Fewest transfers Boolean (optional)
8. Departure and/or arrival time (optional)

The software will only need a starting and ending location in combination with the MBTA data in order to generate a route. We will be using two methods to get our data from the MBTA website. The MBTA Stop data will be parsed by the Jackson Java JSON processor which will convert the data into a Plain Old Java Objects (POJOs). We believe that this will be the easiest data structure to manipulate for these objects. For example, here is a sample set of JSON data from the Jackson Parser website:

```
{
        "name" : { "first" : "Joe", "last" : "Sixpack" },
        "gender" : "MALE",
        "verified" : false,
        "userImage" : "Rm9vYmFyIQ=="
}
```

The above data would be parsed by the Jackson JSON processor into this:

```
public class User {
public enum Gender { MALE, FEMALE };
```

```
public static class Name {
private String _first, _last;
public String getFirst() { return _first; }
public String getLast() { return _last; }
public void setFirst(String s) { _first = s; }
public void setLast(String s) { _last = s; }
}
private Gender _gender;
private Name _name;
private boolean _isVerified;
private byte[] _userImage;
public Name getName() { return _name; }
public boolean isVerified() { return _isVerified; }
public Gender getGender() { return _gender; }
public byte[] getUserImage() { return _userImage; }
public void setName(Name n) { _name = n; }
public void setVerified(boolean b) { _isVerified = b; }
public void setGender(Gender g) { _gender = g; }
public void setUserImage(byte[] b) { _userImage = b; }
}
```

This data structure can easily be applied to the MBTA Stop data, and we chose it because it's a structure that is natural to the Java language. We will take the provided list of stops in .csv format, then convert it to JSON and import that data.

For our train data, we will still be using the Jackson parser, but this time we will be using the raw data binding to get exactly the information we need and store it how we want. This gives us more flexibility in how we acquire data from the MBTA website.

In order to actually calculate the routes through the list of stops we will be using Depth First Search (DFS). Because there is precisely one path between any two stops on the MBTA, we can use a simple algorithm like DFS. If the MBTA was more complicated, or involved multple possible routes between stops, we would need to use a more intensive option like Dijkstra's. This algorithm finds the shortest path between the given source and target. The basic idea is that the algorithm has a list of nodes that connect the source to the target. The algorithm finds a path that works and then goes node by node to see if there's a shorter connection. If it finds a shorter path then it sets that as the new path to return. We will have to adapt this algorithm to deal with the advanced search settings so that it can find routes based on variables such as earliest arrival or departure. This algorithm is easy to implement and can be easily modified so it is a good choice for use in the Trip Planner. Nodes and edges will be represented by a hashmap of stop IDs to a list of the stop IDs of the stops they connect to. The output of this algorithm should be a list of the paths that satisfy the search parameters.

Ordered lists of stops will be handled by calculating and combining the routes between each consecutive stop on the list. Because the list is ordered, the start and end points are specified so we do not need to worry about computing all possible routes between all stops.

Forming routes on unordered lists is more complicated. For this we will compute all possible orders for the stops in the list we are given. Then we will compute an ordered route for each of those possibilities and take the best one (shortest or best timing fit). This should not be a problem because of the relatively small number of stops in the MBTA, and the absence of loops in the system. The downfall of this is that our algorithm will run in factorial time, and thus could run slowly for large unordered lists. Unordered lists will be the slowest search method, because it is doing many time the work of the other two methods.

The TripPlanner class willhave the functions for the basic running of the program, such as calling methods to pull live data on a timer, moving informtation into the route planning graph.

2. Class Structure

The class structure for the application is based on the MVC architecture. The TripPlanner class, represented in our class diagram in FIgure 1 by *Application* is the application layer which runs the data classes of the model layer, the View class (view layer) and the TrainGraph class (controller layer). The View class controls displaying everything on the screen and handling user input.

The TrainLine class instantiations shall contain relaventin formation about a specific MBTA lines. Each line will consist of a list of all of the trains on that line. The TrainLine objects will be updates through the TripPlanner application timer, so that all train lines are updated at the same time.

The Train class will have all information relevant to one train on a line of the MBTA, including location information (description, latitude, and longitude), direction, and a list of Predictions of when the train will arrive at the next stops.

The Prediction class will give information about how soon a train will reach a Stop. This class will have as attributes the train it is predicting for, the stop it is predicting for, and the predicted time in seconds.

The Stop class will list the stops it is next to on the Line for route planning, Stops will be for a specific line of the T, so transfers between lines would act like going between stops. For example, the State Street stop would be represented by both the StateStreet_Blue and StateStreet_Orange stops, who would be listed as connecting to each other. Each stop on a line will be represented once, and direction will be handled by the trains if it is needed.

The TrainGraph object, represented in the class diagram by Route, handles calculations and sorting based on user input and the live data that is stored within the Subway object and in the graph itself. It is in charge of calculating routes between points, aggregating routes for a list of stops and sorting results based on user criteria.

Class Diagram

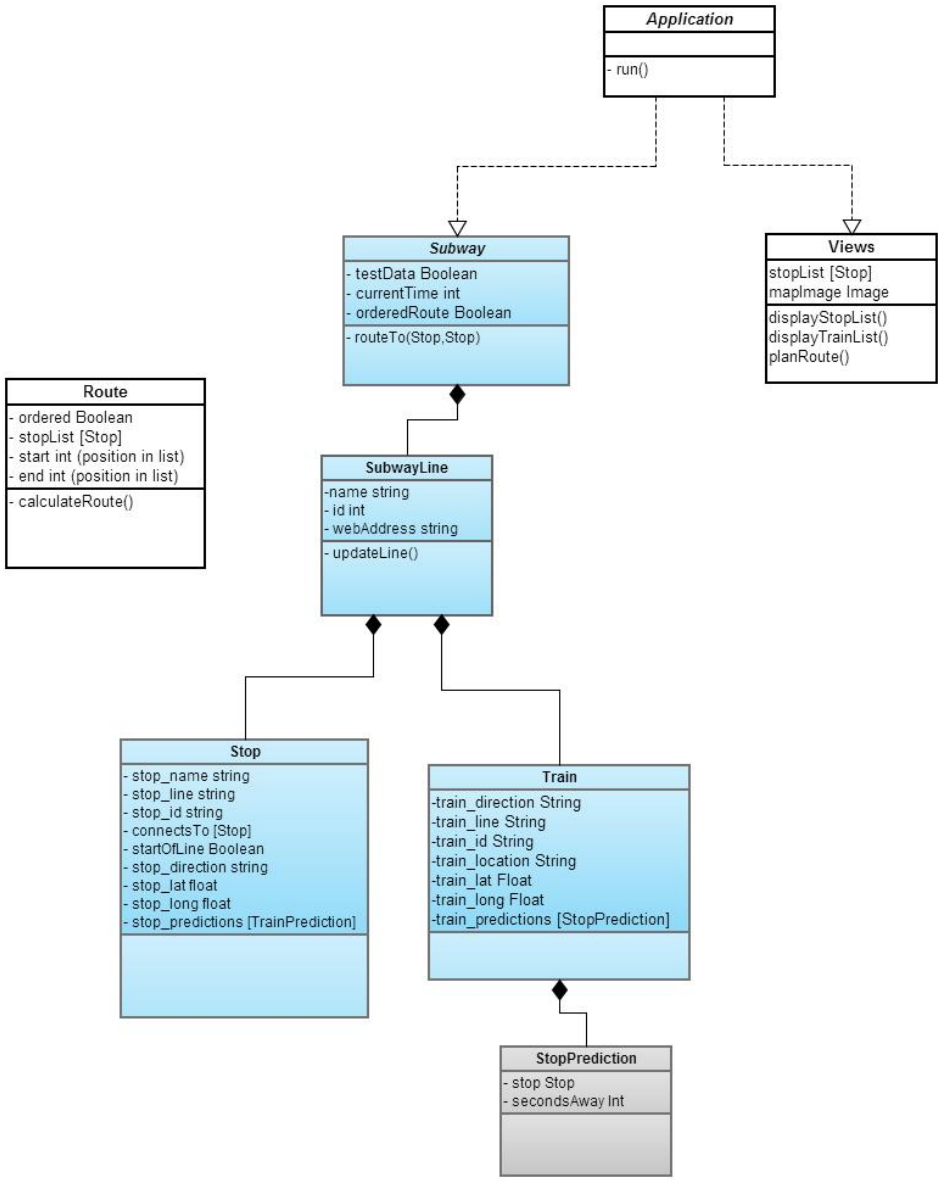

Figure 1