

Automated Train Management System (ATMS)

Dr. Phillipa Bennett

Contents

1 Credits	3
2 Introduction to ATMS	4
3 Requirement for ATMS	5
3.1 Decompositions for P1 and P2	6
4 Classes	8
4.1 Initial Design	8
4.2 Packages	8
4.3 Standard UML and Java Equivalences	8
4.4 Enumerations	10
4.4.1 Example for <code>SystemStatus</code>	10
4.5 Imports	11
5 Tasks for P1	13
6 Addendum to Part 1	16
6.1 October 24, 2022	16

List of Figures

1 An example map of routes in the London Underground	7
2 Initial Design for ATMS for P1	9
3 Enumerations.	10

Listings

1 Java code for the <code>SystemStatus</code> enumeration	10
---	----

1 Credits

The ATMS is an adaptation of the *The Automated Train Management and Control System (TMCS)* first published by Dr. Robert France.

Dr. Ilenius Ildephonse (UWI, Five Islands) provided the rendering of the London Underground (underground railway stations in London).

2 Introduction to ATMS

An Automated Train Management and Control System (ATMS) simulator is to be developed. The ATMS is responsible for moving trains across a railroad system and for controlling traffic signals that determine whether a train should stop or proceed. Train movement is controlled by the system, but there is an engineer onboard each train that can force a train to stop, and to restart after a forced stop.

The ATMS is intended to evolve into a next-generation smart public transportation system that efficiently and cost- effectively move people within and across cities and their suburbs. You will develop the software as a simulation, that is, rather than the ATMS receiving input events from a real-world railroad system with real trains, your ATMS will read input events from an input event simulation file.

In addition, the ATMS implementation will execute simulation loops where each simulation loop is assumed to take exactly 1 time unit to execute. This avoids the use of clocks in the implementation.

3 Requirement for ATMS

The railroad system on which trains will run consists of stations and routes between the stations. A route connects exactly 2 stations. One station is the startStation for the route and the other is the endStation for the route. Each route allows travel in only one direction (from the startStation to the endStation). There can be more than one route between any two stations. Each route is further divided into segments. Only one train can be in a segment at any time. Each segment has a traffic light at the end of the segment. This light determines whether a train can enter the next segment or not. If the light is red then the train is stopped (i.e., it cannot proceed to the next segment). If the light is green then the train can proceed to the next segment. The light is green only when there is no train in the next segment. If there is a train in a segment then the light in the previous segment must be red. Segments are contiguous, that is, when a train leaves a segment it is entering the next segment.

Routes and stations may be closed for maintenance or because of the occurrence of some incident that poses a threat to train passengers.

The ATMS is used to move trains across the railroad system, and controls the traffic lights in each segment. At the start of a journey a train registers its journey with the ATMS. A journey consists of:

1. a journey start time; this time is specified as the number of time units from the time that an approval is received; e.g., if the journey start time is 2 time units then the train will start its journey 2 time units after the journey is approved; the start time cannot be 0,
2. a sequence of segments with a journey start station (start station of the first segment) and a journey end station (end station of the last segment); in a roundtrip journey the start and end journey stations are the same, and
3. a sequence of stations at which the train will stop (the train will stop at the stations in the order given; a stop station must be on one of the journey segments); the journey start and end stations are not included as stop stations – if a list of stop stations is not indicated, the train will stop at all the stations along its route.

During registration, the ATMS checks whether a train's journey is valid, that is, it checks that the journey does not traverse any segments or stop at stations that are currently closed, or stop at stations that are not on the stated journey routes. If the journey is valid then the ATMS approves the train for the journey. A submitting train cannot proceed on a journey until it submits a valid or verified journey.

When a train enters a segment, the light in the previous segment is set to red, and the light in the segment preceding the previous segment (if any) is set to green by the system. In a station, there is one platform (station segment) for each incoming route – this means that only one train can be in a segment at any time.

Each route, segment, or station must have a unique name. Note that a uniquely identified station can appear on more than one routes.

Figure 1 provides a useful illustration of routes, segments, and stations. For example:

1. West Ruislip, North Action, Baker Street, and Bond Street are stations;
2. North Action to Notting Hill Gate is a segment; and
3. the sequence of stations:
 West Ruislip \rightarrow North Action \rightarrow Notting Hill Gate \rightarrow
 Bond Street \rightarrow Liverpool Street \rightarrow Epping
and
 Baker Street \rightarrow Liverpool Street \rightarrow Westminster \rightarrow Notting Hill Gate \rightarrow Baker Street
are routes with the latter route being a round trip.

3.1 Decompositions for P1 and P2

Part *P1* of this project will implement the *Train*, *Route*, *Segment*, *Station*, and *TrainManagement* classes. Part *P2* will add the Simulator functionality.

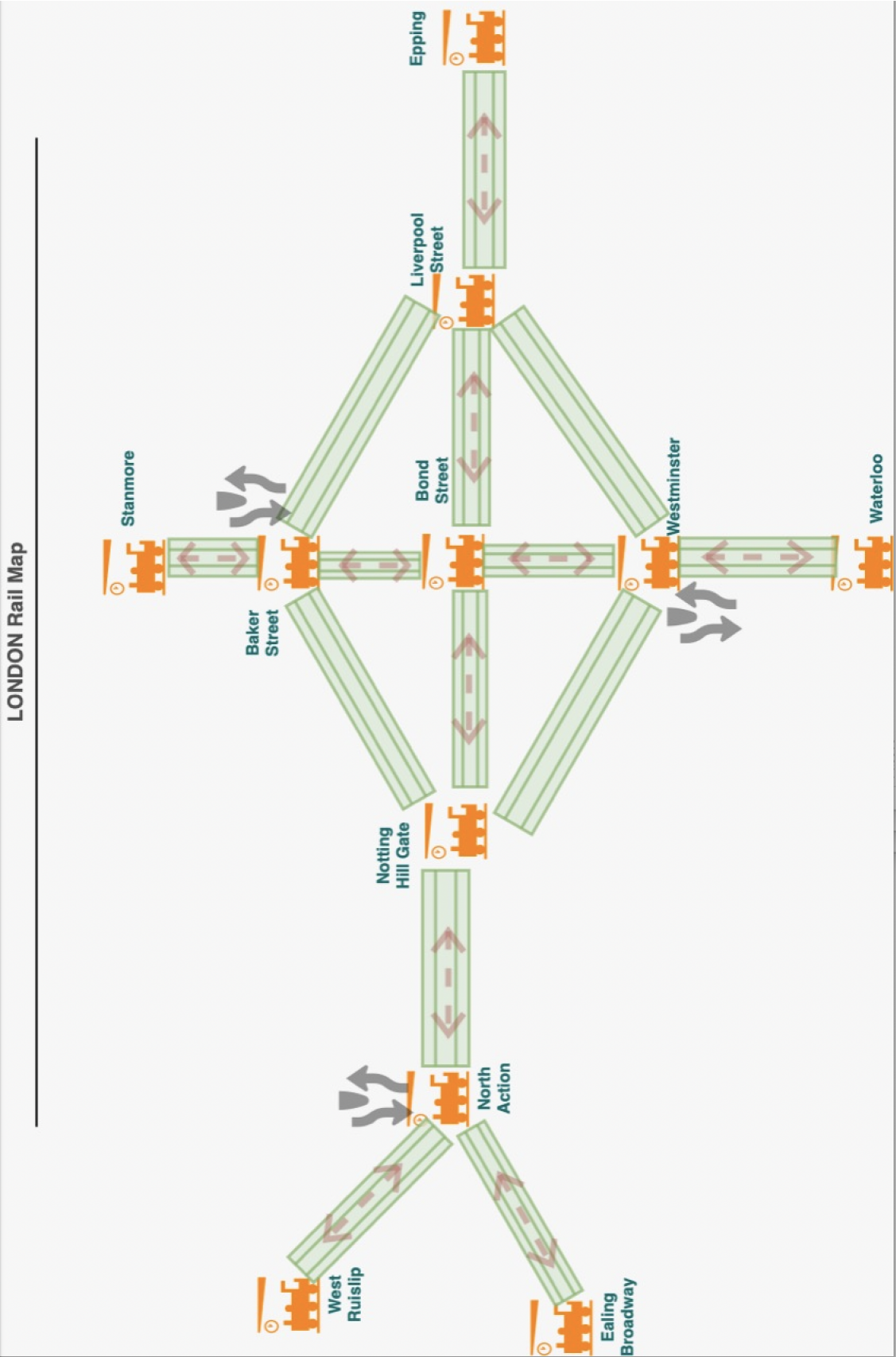


Figure 1: An example map of routes in the London Underground

4 Classes

4.1 Initial Design

The initial design for the ATMS is shown in the UML class diagram in Figure 2.

While not represented, all the attributes should have private visibility and all the methods should have public visibility; no other attributes or public methods should be added to the classes.

A starting specifications is also provided with some methods already implemented – in it you will see other accessors, mutators, and helper methods in each class. Unless otherwise indicated/advised/instructed:

1. do not change the visibility modifier or the return types on these methods;
2. do not change the implementations of these methods; and
3. except for the `Station` class the `compareTo()` method is included to be able to easily sort the objects when they are in a collection; therefore you should not rely on these methods when checking that one object is equal to another.

4.2 Packages

All classes, including enumerations, are to be placed in a package called `p1`; this means that the first line of each file must have the following line:

```
package p1;
```

If a method should search a collection and returns a value from the collection, if the value is not found, return a null reference for object types and -1 for integer types.

4.3 Standard UML and Java Equivalences

Figure 2 uses standard UML Class Model declarations and types; their equivalences in the Java language are as follows, the UML:

1. *Integer* type becomes the any of the integer primitive types in Java and you choose the type based on the range of values that are required in your domain;
2. *UnlimitedNatural* type becomes the any of the integer primitive types in Java with the added constraint of being non-negative;
3. *Real* type becomes the any of the real valued primitive types in Java you choose the type based on the range of values and the precision required in your domain;
4. *Boolean* type becomes the `boolean` primitive type in Java; and
5. *OrderedSet* type or *ordered* on a role denotes an ordering/sequencing of the values and for this project we will use the `ArrayList` type in Java – for example the UML *OrderedSet(Route)* would be the `ArrayList<Route>` in Java.

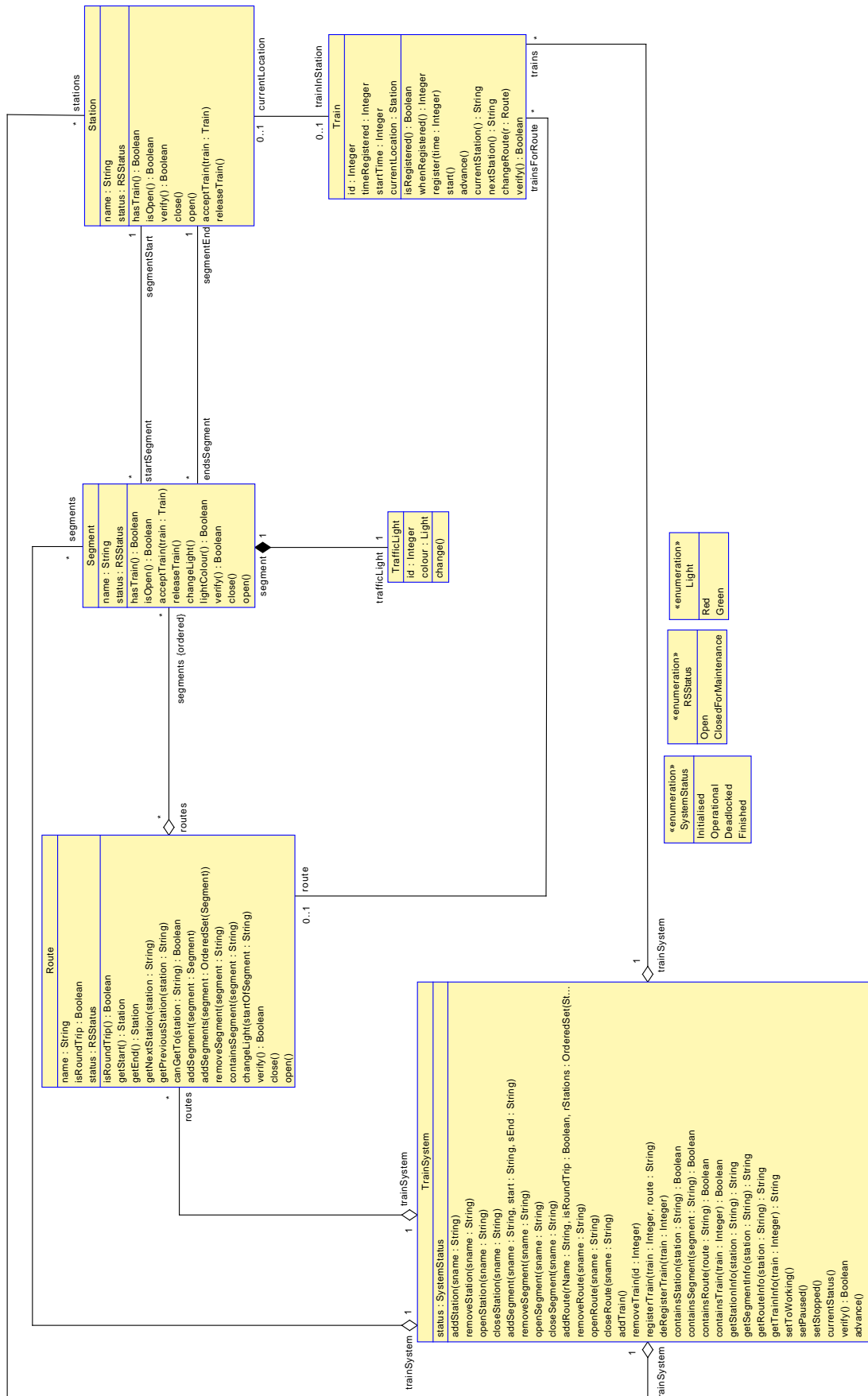


Figure 2: Initial Design for ATMS for P1

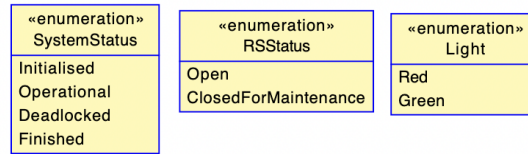


Figure 3: Enumerations.

In addition to the attributes in the classes, UML diagrams (should) show attributes that are collections as roles on the association lines. These collections will be implemented as `ArrayList` of the associated type in java. Not all the roles will be used in this project – the ones used are included in the starting specifications.

You should keep in mind that `ArrayLists` allow duplicates but the collections associated with the `TrainSystem` class should not have duplicates, whether by aliases or by using non-unique identifiers.

4.4 Enumerations

The enumerations are (again) shown in Figure 3. Each literal in each enumeration type will have a description.

4.4.1 Example for `SystemStatus`

As an example, the definition for the `SystemStatus` is given in Listing 1 where both a constructor and an accessor (getter) method are included.

Listing 1: Java code for the `SystemStatus` enumeration

```

package pl;

public enum SystemStatus {

    Initialised("System is Initialised"),
    Operational("System is Operational"),
    Deadlocked("System is Deadlocked"),
    Finished("No More trains!");

    private String description;

    SystemStatus(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
}
  
```

All the other enumerations are similarly defined.

4.5 Imports

The following imports of class libraries may be useful to you:

1. `import java.util.ArrayList;`

This class library is mainly used to store collections. See

<https://docs.oracle.com/javase/8/docs/api/?java/util/ArrayList.html>
for its JavaDoc documentation.

2. `import java.util.Arrays;`

This class library is mainly used to create or sort collections – your `int compareTo(...)` method in the associated classes must be implemented correctly for the sort to produce correct results.

As an example, suppose you have the following variable defined as:

```
ArrayList<String> stringList;
```

and you want to initialise it with a list of strings, then you may use:

```
new ArrayList<String>(Arrays.asList(new String[] { "Cross Roads", "Half Way Tree", "Downtown" }));
```

In another example, suppose you have the following:

```
ArrayList<RStation> stations;
```

where the `Station` class implements the `Comparable<Station>` interface, then to sort the `stations`, you may use the following statements:

```
if (stations.length() > 0){  
    Object[] z = stations.toArray();  
    Arrays.sort(z);  
    stations = new ArrayList(Arrays.asList(z));  
}
```

Alternatively, you could use the other form of the `toArray()` method as follows:

```
if (stations.length() > 0){  
    Station[] acc1 = stations.toArray(new Station[0]);  
    Arrays.sort(acc1);  
    List<Station> ss = Arrays.asList(acc1);  
}
```

and use `ss` in your computations.

See

<https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html>
for the JavaDoc documentation for the `Arrays` class library.

3. `import java.util.Iterator;`

You may use the class library in much the same way as you would iterators like `Scanner` objects.

As an example, suppose you have the following variable defined as:

```
ArrayList<String> stringList;
```

you may use:

```
Iterator<String> sgs = stringList();  
while (sgs.hasNext()){  
    // code that includes the use of sgs  
}
```

to process the elements of the collection.

5 Tasks for P1

1. *This task is worth 30% of the marks.* Using the document at <https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html> as a guide, provide Javadoc comments for all of the methods (including the methods in the enumerated types) – at a minimum you must add comments to:
 - a) describe the method;
 - b) describe each of the parameters (use the `@params` keyword for each parameter); and
 - c) if the method has a return type, give a description how to interpret or use the return type (use the `@returns` or `@return`).

Keep in mind that:

- a) Javadoc comments are enclosed using a double asterisk at the beginning of the comment, like this

```
/** This is a Javadoc comment */
```

instead of a single asterisk in

```
/* This is a regular comment */
```

for other kinds of comments that are put in a Java file.
- b) writing the Javadoc will help you to go through and understand the starting code.
- c) when you are reviewing that code, keep in mind that given the following declaration:

```
ArrayList<Station> stations
```

the statement

```
stations.forEach(s-> System.out.println(s))
```

is an equivalent form of:

```
for (Stations s: stations)
    System.out.println(s)
```

2. *This task is worth 70% of the marks.*

- a) implement the *verify()* method in each of the classes in which it appears; in the
 - i. **Station** class the method should ensure that its is valid, i.e., name is not null or an empty string, including being filled with whitespace.
 - ii. **TrafficLight** class the method should ensure that its colour is not null.
 - iii. **Segment** class the method should ensure that:
 - A. its name is valid;
 - B. its rafficl light, segment end, segment start are verified;
 - C. its segment start and segment end are not the same, whether by aliases or by non-unique identifiers; and
 - D. if the segment is open then both segment start and segment end are also open, also vice versa.
 - iv. **Route** class the method should ensure that:
 - A. its name is valid;
 - B. its segments is non-empty;
 - C. if it is a round trip that its start and end station are the same;
 - D. if it is not a round trip that its start and end station are not the same;
 - E. its segments are properly sequenced;
 - F. there are no duplicate segments, whether by aliases or by non-unique identifiers;
 - G. each of its segments is verified; and
 - H. except for round trips, there are no loops in route, for round trips the loop can exist because the start station and the end station is the same.
 - v. **Train** class the method should ensure that the route is verified and the time registered is greater than 0.
 - vi. **TrainSystem** class the method should ensure that each object in the *stations*, *segments*, *routes*, and *trains* are verified and that there are no duplicates objects in any of the collections – keep in mind that duplicates may exists because of aliasing or non-unique identifiers.
- b) in the **Route** class implement the:
 - i. `private ArrayList<Station> getStationList()` method – it should return the sequence of stations in the route, i.e., first station to last station; and
 - ii. `public String getPreviousStation(String station)` method.
- c) implement the **TrainSystem** class – you should
 - i. notice that the *toString()* is already implemented – do not change its implementation; and

- ii. keep in mind that what is checked in `verify()` method should constrain your implementation of the methods in this class.
- d) do not implement/change the *advance()* method in any of the classes; they will be implemented in *P2*, however you should start thinking about how they could be used to move the trains during simulation.

For P1, we will be testing the methods in the classes, therefore a user interface where values are accepted from the user is not required for submission. However, you can write one to test your system.

6 Addendum to Part 1

Corrections and clarifications after the initial publishing of this document will be added here.

6.1 October 24, 2022

Initial Document.