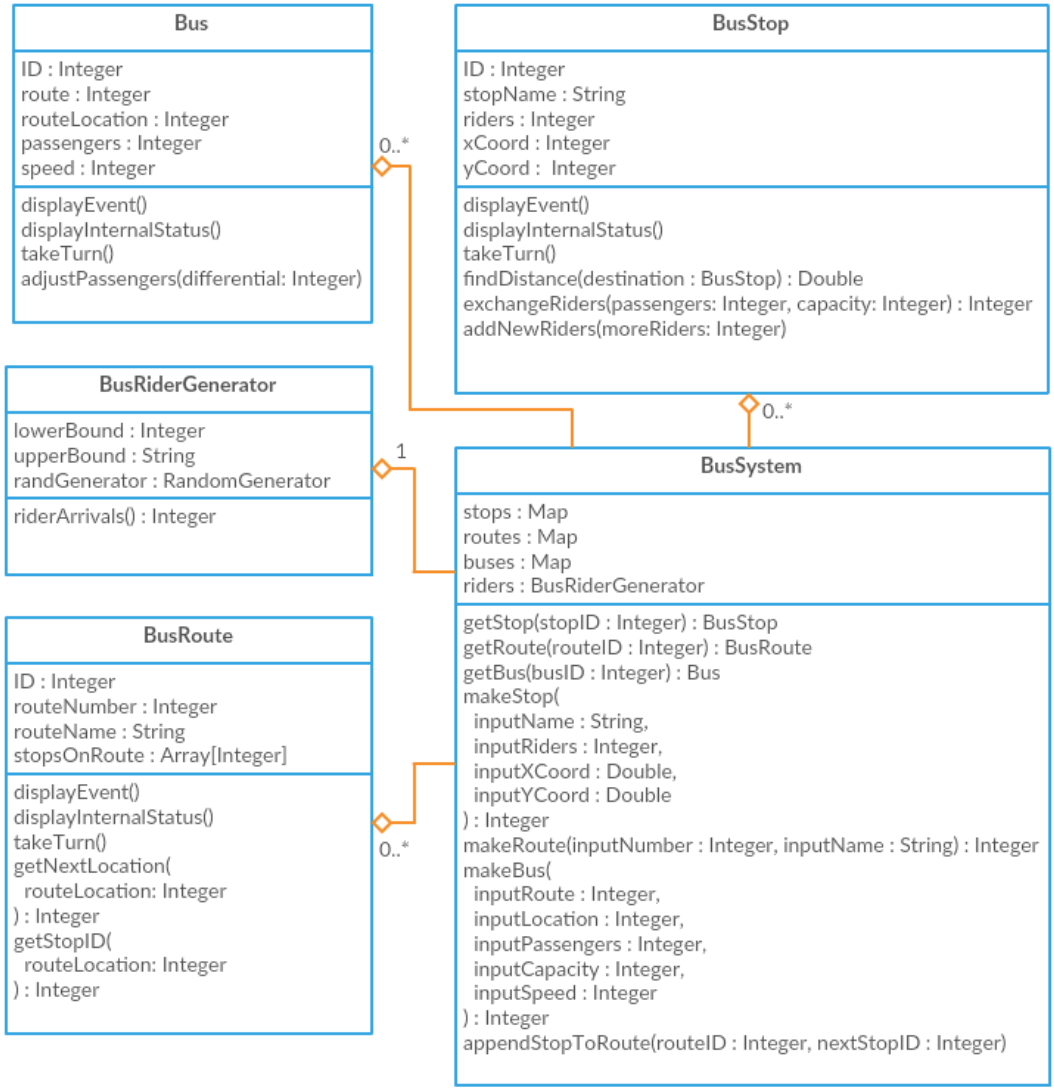
Daniel Adelberg
OMSCS Course 6310, Spring 2018
February 8, 2018

Introduction

This paper is an analysis of the Mass transit Simulation System (MTS), a java program included in assignment 3, OMSCS' Course 6310. I use UML 1.4 to diagram the system, using a Class Diagram, an Object (Instance) Diagram, and several Sequence Diagrams. Each Diagram type may be broken up across several images. I will preface each image with an explanation of any non-standard notation.

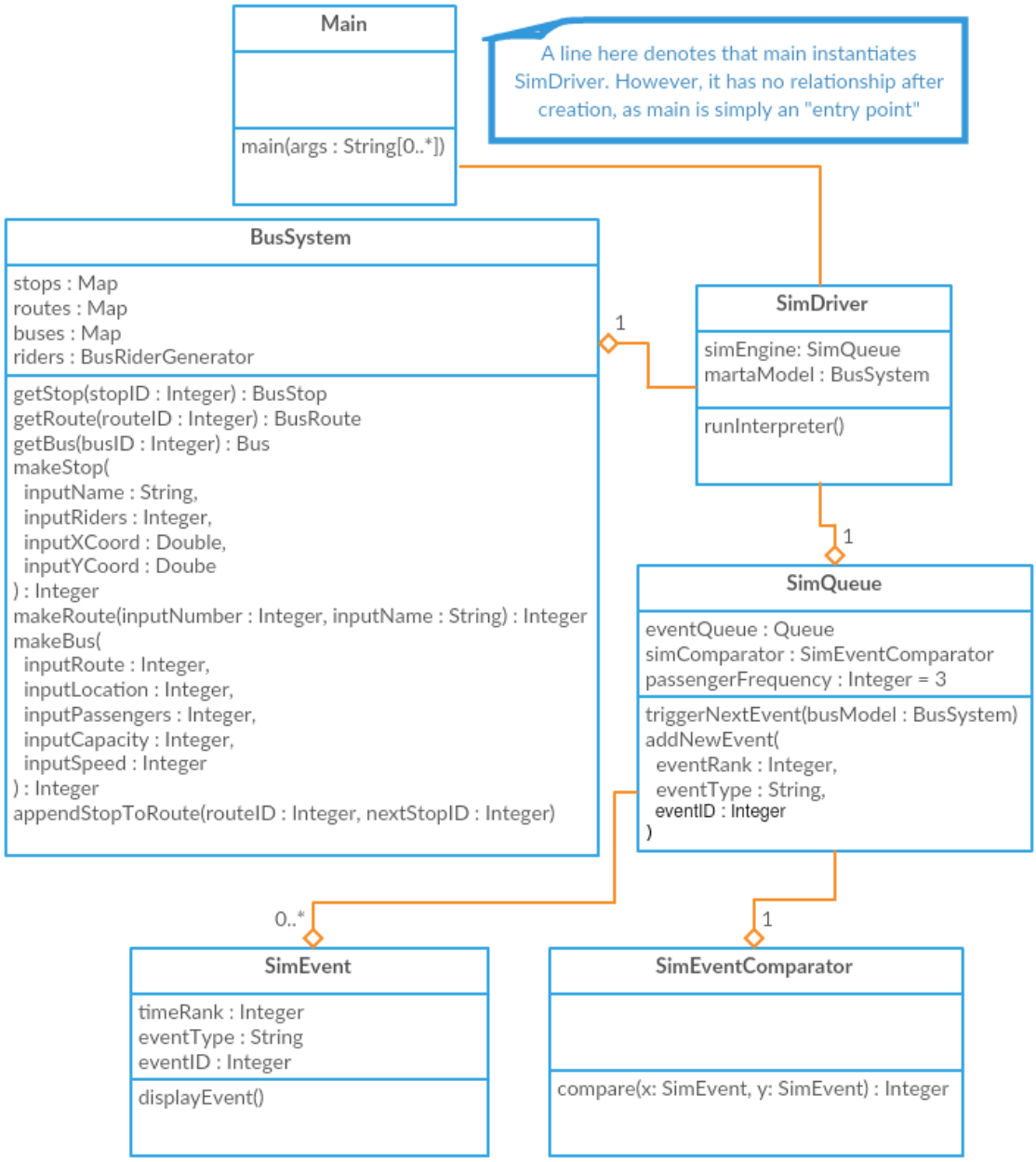
Class Diagram

To start, we consider the program using a class diagram in two parts.



Part 1: BusSystem and its constituents

Above, I have modeled the BusSystem and its constituents. Note the multiplicity of the aggregations: while a BusSystem may have any number of Buses, BusStops, and BusRoutes, it can only have a single BusRiderGenerator. Note also that I have used the types “Double” and “Map”. These can be assumed to be a Real-valued number to a given precision and a set of key-value, respectively.



Part 2: SimDriver, SimQueue, and related immediately related objects

Here I introduce a comment to indicate the meaning of a relationship. I also use an assigment-like syntax in SimQueue to indicate a hardcoded initial value for passengerFrequency.

Object Diagram

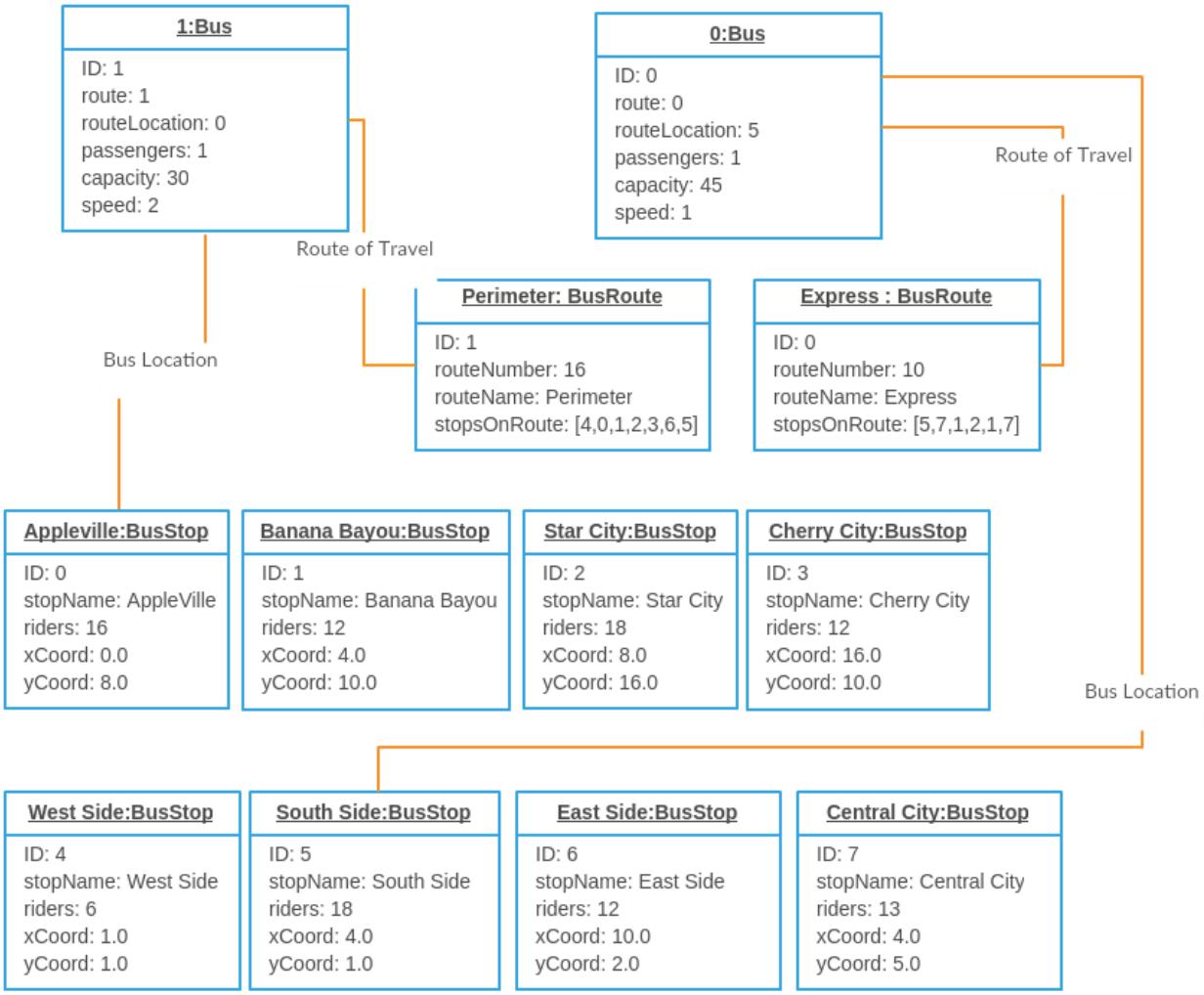
The following Object Diagrams represent the state of the system when ‘system_report’ was sent during this command sequence:

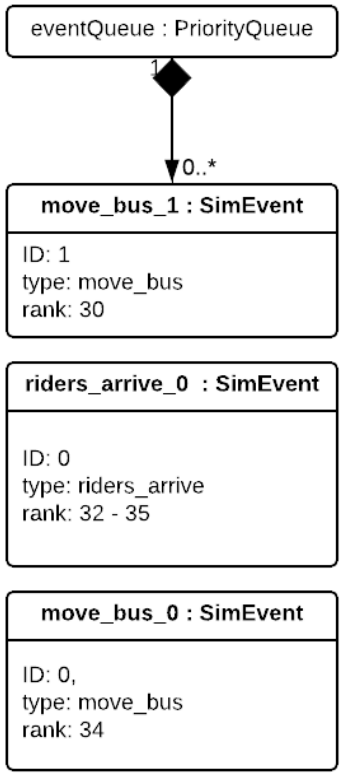
commands.txt

```
1  add_stop,Appleville,5,0.0,8.0
2  add_stop,Banana Bayou,0,4.0,10.0
3  add_stop,Star City,0,8.0,16.0
4  add_stop,Cherry City,0,16.0,10.0
5  add_stop,West Side,0,1.0,1.0
6  add_stop,South Side,0,4.0,1.0
7  add_stop,East Side,0,10.0,2.0
8  add_stop,Central City,0,4.0,5.0
9  add_route,10,Express
10 add_route,16,Perimeter
11 extend_route,0,5
12 extend_route,0,7
13 extend_route,0,1
14 extend_route,0,2
15 extend_route,0,1
16 extend_route,0,7
17 extend_route,1,4
18 extend_route,1,0
19 extend_route,1,1
20 extend_route,1,2
21 extend_route,1,3
22 extend_route,1,6
23 extend_route,1,5
24 add_bus,0,0,0,45,1
25 add_bus,1,0,0,30,2
26 add_event,1,move_bus,0
27 add_event,1,move_bus,1
28 add_event,2,riders_arrive,0
29 step_multi,20
30 system_report
31 quit
```

The following Object Diagram shows the state of the program after 20 steps are made in the program, where the state of the program was determined by the sequence of commands issued above. Note that I have not drawn the busStop - busRoute relationships. There relationships are quite numerous, but unchanging during the program run. Each integer in stopsOnRoute corresponds to the ID field of a given busStop. In this way, one can observe the relationship implicit in the diagram without polluting the diagram with lines representing static connections.

Additionally, I have not declared the types for attribute primitives. This is because they are designated in the class diagram above, and add cruft to the object diagram. My assumption here is that the reader will reference the class diagram to determine types for attributes, if that is a source of confusion for said reader.



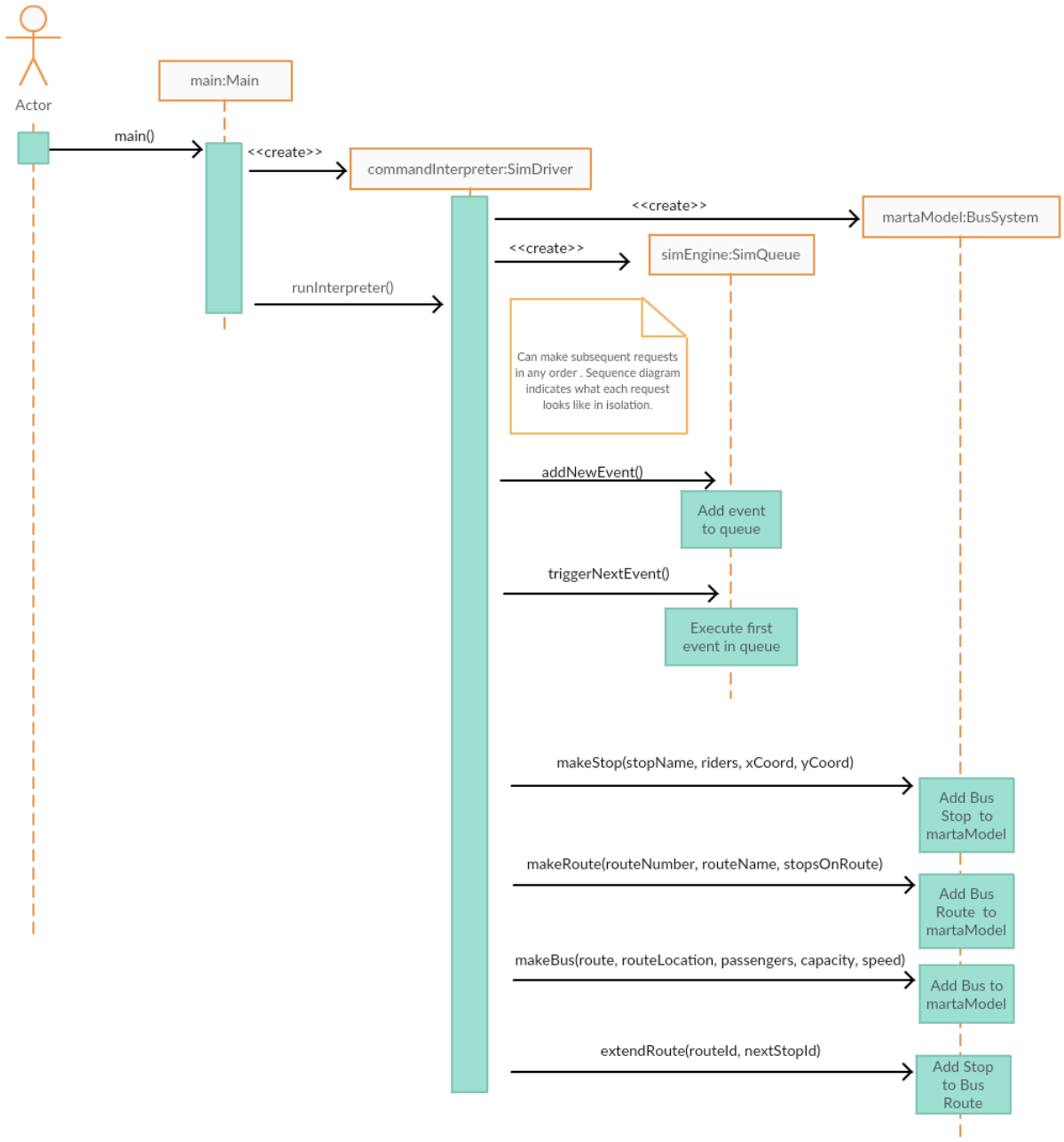


Object diagram of eventQueue

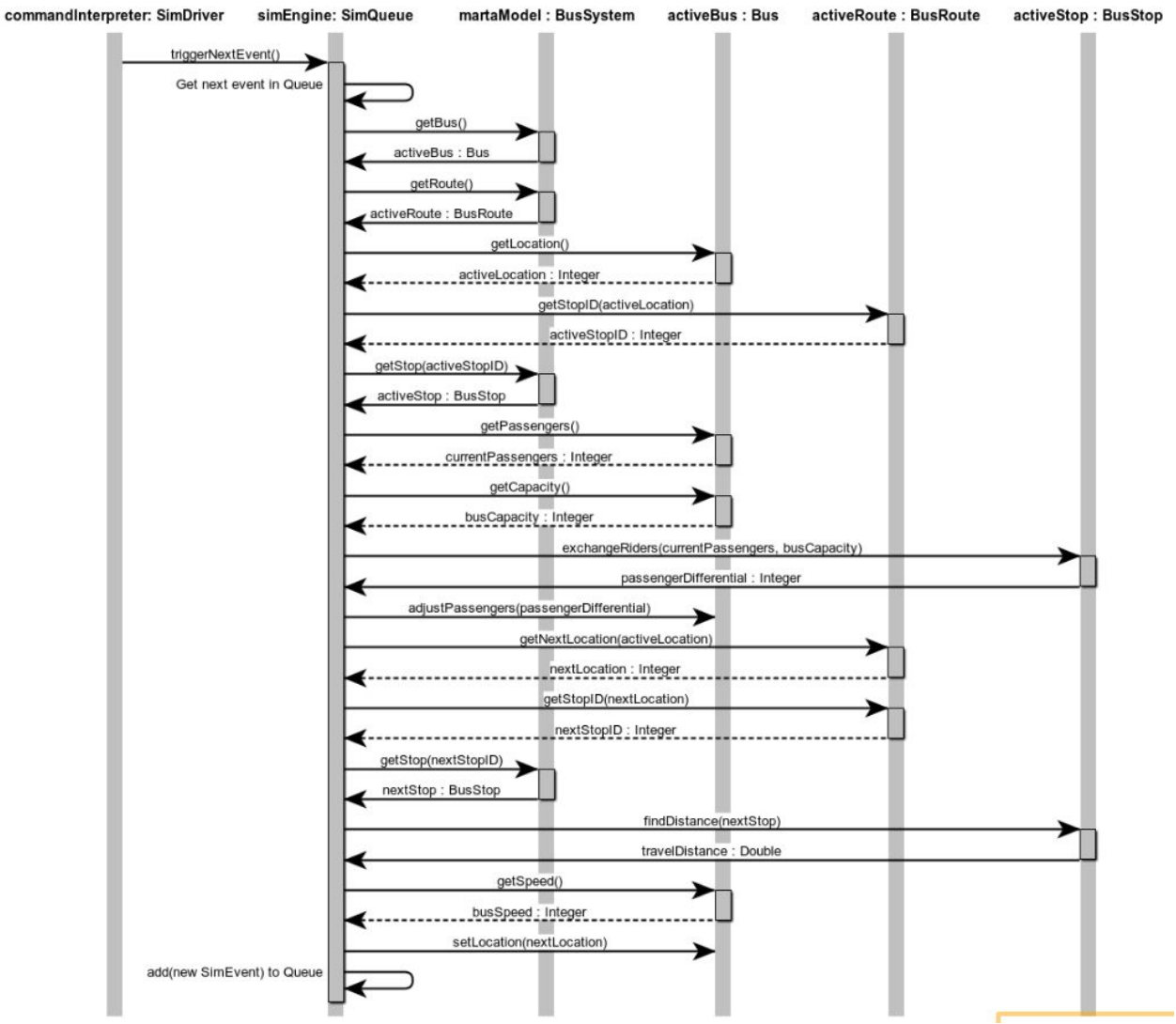
The event queue can also be diagrammed. On the right I have diagrammed the state of the event queue after 20 steps have been taken. Note that the eventQueue has 3 events in it, and that the ‘riders_arrive’ event has a range for its rank. This is because calculating a rank for a ‘riders_arrive’ event uses a non-deterministic function call (random). Since the system is not deterministic, I provide a range of possible values to indicate the range of possible ranks for the SimEvent instance.

Sequence Diagrams

Last, I describe the program using a Sequence Diagram. The diagram is split into several parts, reflecting different aspects of the system. Ordered by appearance, these diagrams cover system initialization and initial message passing to initialize the queue, the “move_bus” action, the “riders_arrive” action, and finally the mechanics of adding a new event to the simulation queue.

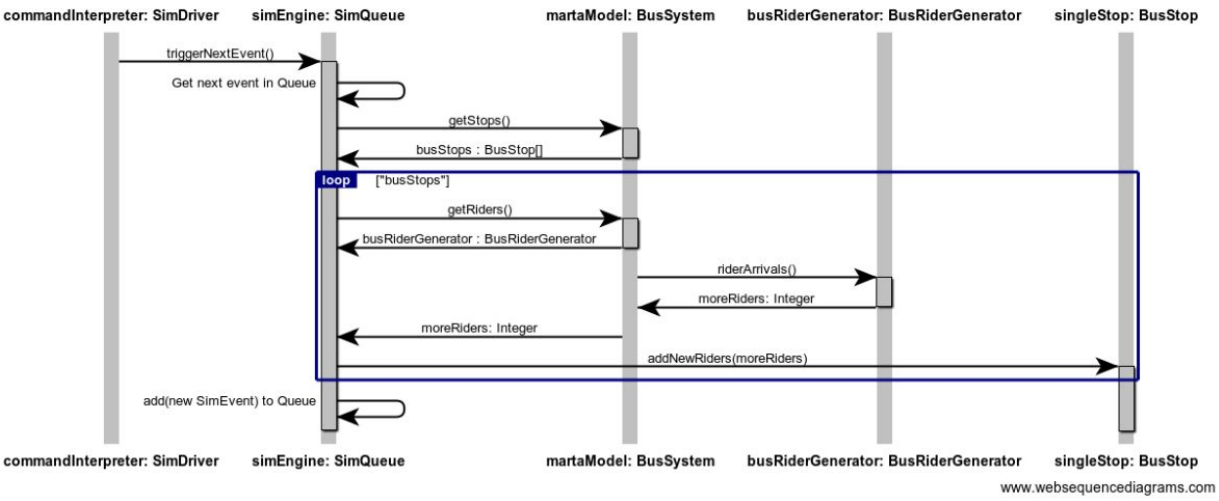


Note that the above sequence diagram describes the initialization of the program. The events sent by the commandInterpreter (either ot the simEngine or martaModel instances) could be in any order. This is because the commandInterpreter is looping over over any arbitrary input commands.



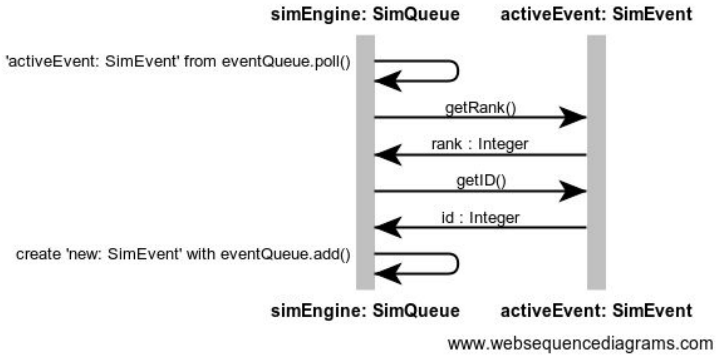
Modeling the actions taken when a “move_bus” event is processed

Note that the details of new SimEvent self-call at the end of this diagram does not elucidate calls to activeEvent. This is intentional: the calls to activeEvent (getRank() and getID()) are tangential to the “move_bus” sequence. They pertain more to the mechanics of setting a new event than to the “move_bus” event itself.



Model for processing a “riders_arrive” event

Last we have the eventQueue interaction with the current activeEvent. This diagram explains the calls made when a SimEvent is created and then added to the eventQueue.



Closing Thoughts

The diagrams above serve to document the core aspects of the program from several different “viewing angles”. As such, they do not seek to model everything the program is doing. Writes to standard-out are universally omitted. Trivial getters and setters, constructors, and other ‘intuitive’ aspects of the program are similarly omitted. These diagrams should be used to get a sense of the program from an architectural standpoint. Implementation details are left to the program developer.