CS6310 – Software Architecture & Design
Assignment #2 [100 points]: Mass Transit Simulation – Requirements Analysis & Design (v3)
Fall Term 2018 – Prof. Mark Moss

Submission

- This assignment must be completed as an individual, not as part of a group.
- You must submit:
  - (1) a UML Class Diagram named **uml_class_diag.pdf**; and,
  - (2) a UML Sequence Diagram named **uml_sequence_diagram.pdf**.
- You may submit multiple diagrams if needed – for example, if the diagrams are so large that dividing them makes it easier to present them in a more readable format. In that case, you may submit the multiple files with reasonable name extensions such as **uml_sequence_diagram_1.pdf**, **uml_sequence_diagram_2.pdf**, etc. or **uml_sequence_diagram_move_bus.pdf**, **uml_sequence_diagram_reset_sim.pdf**, etc.
- Formats other than PDF must be pre-approved an Instructor or TA.

- You must notify the Instructors and TAs via a private post on Piazza BEFORE the Due Date if you are encountering difficulty submitting your project. Send the message to "Instructors" when posting your message in Piazza, as opposed to addressing a message to just one individual, to ensure that it is acknowledged as quickly as possible.
- You will not be penalized for situations where Canvas is encountering significant technical problems. However, you must alert us before the Due Date – not well after the fact. You are responsible for submitting your answers on time in all other cases.
- Please consider that uploading files to Canvas might occasionally take a long time, even in the case of seemingly "relatively small" submissions. Plan accordingly, as submissions outside of the Canvas Availability Date will most likely not be accepted. You are permitted to do unlimited submissions, thus we recommend you save, upload and submit often. You should use the same file naming standards for the (optional) "interim submissions" that you do for the final submission.

Scenario

The application that we will be working with is a Mass Transit Simulation (MTS) system. The prototype application that has been provided simulates the interaction of buses moving along a route of stops, and allowing passengers to board and later depart the bus at different stops. Your requirement for this assignment is to create Unified Modeling Language (UML) Class Diagram and Sequence Diagrams that accurately reflect the concepts of the problem space. The clients have provided you with the prototype to help clarify their requirements and intentions. You must interpret the client's descriptions of the problem space, and requirements, as given below and in other (later) discussions.

Disclaimer

This scenario is inspired by the Metropolitan Atlanta Rapid Transit System (MARTA) Expansion Project as part of the Georgia Tech Serve-Learn-Sustain (SLS) Program, and we will leverage some of their data and other resources. Any other similarities or differences between this scenario and any of the

programs at Georgia Tech programs are purely coincidental. More information on the MARTA Expansion Project can be found at:
http://serve-learn-sustain.gatech.edu/marta-expansion

Deliverables

UML Diagrams are normally divided into three fundamental categories: Functional, Structural and Behavioral. For this assignment, you must analyze the MTS System that has been provided, and then submit UML Diagrams that accurately reflect the system as described by the clients. You must submit the following items:

(1) Class Diagram [50 points] – You must generate a Class Diagram that includes (at a minimum) classes, attributes with basic types, methods, and relationships with proper cardinalities that accurately reflect the problem space. To make your class diagram more readable, you may omit very basic "setters and getters" [e.g. set_() and get_() based operations and methods], especially those of the one-line variety that only access and/or modify an object's basic attribute. Your UML Class Diagram must match the same level of detail as shown in the Udacity videos.

(2) Sequence Diagram [50 points] – You must generate a Sequence Diagram (sometimes referred to as an Event Diagram) that reflects the actions performed in the "Move Next Bus" command. Your sequence diagram must be consistent with the class and object structures that you designed in part (1), where the sequence elements reflect the messages sent between objects in your model, along with conditional and iteration structures as required. If you can fit everything into one diagram, great; if not, then use multiple diagrams as required to ensure that the results are readable.

MTS System Description

The Mass Transit Simulation (MTS) application must implement a discrete-event simulation that allows buses to travel along different routes while transporting riders to different stops. The description below gives a brief, high-level overview of how these concepts are interpreted by the clients. The task of the course long project will be to design and develop this application, which will include the current description of the problem space and system, along with new requirements. For this assignment, however, your goal is to develop the "to-be" design artifacts based on the client's current description of the problem space.

- **Discrete-Event Simulation:** A discrete-event simulation of a system is characterized by modeling the actions and activities that change the state of the system as a discrete sequence of events in time. There are no changes to the state of the system between events. We will model activities such as buses moving to a new stop, passengers arriving at a stop, etc. as discrete events. Each step of the simulation has two phases, where the first phase involves determining the next chronological event that should be executed. Each event is composed of a program of actions, where each individual action could involve modifying some aspect of the simulation's state and/or creating one or more new events. The second phase involves executing the actions required by that event. If there are two or more events that are "tied" chronologically, then the simulation

system is allowed to act in a non-deterministic manner, and may execute the tied events in any order. Also, the event that has just been executed is normally removed from the pool of candidate events – or, at least its time for the next execution cycle is updated to a new future time.

- **Passengers:** Passengers will represent the people who ride buses from their arrival stop to their destination stop. We will simulate the arrival of passengers at each stop using a random distribution based on the MARTA data that we've collected as part of the SLS program. The data is contained in a database, and is used to generate upper and lower limits for the probability distributions of passenger arrivals and departures for different hours during the day. Quick note: "riders" and "passengers" are both people moving through our simulated transportation system, and there isn't intended to be any significant distinction between the two terms.

- **Buses:** Buses will carry passengers between stops. Buses will travel along a route; and, at each stop, some of the current passengers will get off of the bus, and then some new passengers will get on. Each bus will travel at an average speed, and will also have a capacity to hold a maximum number of passengers. The bus will follow a designated route to determine its next stop.

- **Stops:** Stops are locations where passengers can get on and off the buses. We will not be concerned with passengers getting on or off buses at other locations – only at a valid stop. When a bus arrives at a stop, then a specific sequence of events occur based on the normal actions of some passengers arriving at the stop, different numbers of passengers getting on and off of the bus, and some of the passengers departing the stop. There might also be conditional events that occur based on other actions that occur, or based on changes initiated by the simulation users.

- **Routes:** Routes are lists of stops that determine how buses travel between stops. A bus travels to the stops listed in a route is simple sequence; and, when the bus is located at the last stop in the route list, the bus then "restarts" by going to the first stop on the route list. For example, if a route list has N stops listed [s(0), s(1), s(2), ..., s(n-2), s(n-1)], then the bus currently at stop s(n-1) will proceed to stop s(0) next. A stop can be listed multiple times within a route, and the simulation users will define the routes based on their coverage patterns. Routes will normally be linear (e.g. back and forth along an East-West or North-South orientation) or circular (e.g. perimeter routes around a specific area such as a campus, city or business district), but aren't restricted to such simple patterns.

The client has a very simple prototype system that can be used to get an idea of what is expected. The full application that you will develop will have more functionality. The information below explains that commands for the prototype application, and also lists the commands and functionality that you will need to design and implement as part of this project.

Simulation System Commands

This is a quick guide the MTS prototype that you've been given for demonstration and learning. The simulation has three fundamental aspects: (1) Scenario Definition, which refers to the

commands used to setup the simulation scenario; (2) Simulation Management, which refers to executing the scenario, and to making changes to aspects of the scenario to explore the effects; and, (3) Simulation Monitoring, which refers to capturing and analyzing various aspects of the state of the scenario, such as the number of passengers at a certain stop.

Scenario Definition

The definition category includes commands such as **`add_stop`**, **`add_route`**, **`add_bus`** and **`add_event`**. These commands are used to create the objects in the simulation environment that correspond to the entities described above. The **`extend_route`** command is used to modify a route object by adding a stop identifier to the end of the route list. These commands are saved to a text file that is used when starting the program.

- **`add_stop,<ID>,<Name>,<Riders>,<Latitude>,<Longitude>`**

This command creates a stop object with a given <ID>, <Name>, and an initial number of <Riders>, located on the travel map at the location <Latitude>, <Longitude>.

Each stop must have a unique ID, along with a "user facing" and more descriptive Name attribute. A stop also contains a Riders attribute, which records the initial number of people waiting at the stop to board an available bus. Each stop also contains a Latitude and Longitude attributes that translate to its geographical location on the travel map, and where the Latitude and Longitude correspond to the Y and X coordinates, respectively. The coordinate attributes are used to calculate the distance between stops, which is then used along with the Speed attribute of a bus to determine when the bus will arrive at its next stop.

- **`add_route,<ID>,<Number>,<Name>`**

This command creates a route object with a given <ID>, <Number> and <Name>. The route initially doesn't have any stops.

- **`extend_route,<Route-ID>,<Stop-ID>`**

This command appends the stop designated by <Stop-ID> to the end of the route designated by <Route-ID>.

Each route must have a unique Route-ID, along with more descriptive "user facing" Number and Name attributes, such as "Route 10 – Midtown Express". Most importantly, each route contains an attribute that lists the stops (using the Stop-ID values) covered by that route. The order of the list is important: normally, a bus begins at the stop listed in the first position of the list, and then progresses to each stop as ordered in the given route list. If the bus is located at the last stop on the list, it then goes back to the first stop in the list, and continues the cycle as long as required for the duration of the simulation session. Please note that a route might have the same stop listed twice, which is not necessarily an error. In these cases, the bus is likely traveling along a fixed path in alternating directions – for example, first Northbound, then Southbound, etc.

- **`add_bus,<ID>,<Route>,<Location>,<Initial-Passengers>,<Passenger-Capacity>,<Initial-Fuel>,<Fuel-Capacity>,<Speed>`**

This command creates a bus object with identifier <ID> that travels along <Route>, starting at index <Location> within the <Route> list, and carrying an initial number of <Initial-Passengers> riders. Also, the bus can hold at most <Passenger-Capacity> passengers, and travels along the route at the given <Speed> as measured in statute miles per hour. Also, the bus begins with an amount of fuel needed to travel a distance of <Initial-Fuel> statute miles. The bus, when fully refueled, can travel a maximum distance of <Fuel-Capacity> miles.

Each bus must have a unique ID and travel along a certain Route. To facilitate movement along the route, the bus Route Location attribute refers to an item in the route's list of stops, rather than having the bus refer directly to the stop itself. Also, each bus has a Capacity attribute which limits the total number of passengers that it can carry at any one time, along with a Speed attribute that affects how quickly the bus travels along the route. And for our purposes, the client has stated that they will likely ignore the direct fuel management aspects in the future.

- **`add_event,<Time>,<Type>,<ID>`**

This command creates an event object that will be executed when the simulation reaches "logical time" <Time>. When executed, the system will perform the actions corresponding to the event <Type> using the object designated by <ID>.

Events represent the different types of actions that we will track in our simulation. Each event has a Time attribute, which corresponds to the logical time that this action will happen during the simulation, and is used to organize events in the simulation queue. Each event also has a Type attribute, and an ID attribute which is used to designate which simulation object in the system – bus, stop or route – should be used to execute the associated event. The only event type that we currently have in the system is the **`move_bus`** event, which corresponds to transitioning a bus from its current stop to its next stop based on the distance between the two stops, and the current speed of the bus. The arrival of the bus at its next location also triggers actions involving current passengers getting off of the bus, followed by new passengers getting back onto the bus.

To improve the "synchronization" of events during the simulation, the distance and speed calculations are adjusted with conversion factors so that the rank of an event corresponds to when the event would be executed in minutes from the start of the simulation run. For example, if the event being processed refers to a bus (ID) #17 moving at time 501, and it will take the bus 6 minutes to travel to the next stop, then a new event will be created for the next event for bus #17 with rank 507. See the section below about Calculating Distances and Travel Times for more information about translating the geographical latitudes and longitudes into statute miles.

Other Scenario Configuration Issues

The Scenario Definition capabilities also include an **`add_depot`** command, which is related to fuel management, and leftover from some earlier development on the client's prototype.

- **`add_depot,<ID>,<Name>,<Latitude>,<Longitude>`**

This command creates a special stop object with a given <ID> and <Name> located on the travel map at the location <Latitude>, <Longitude>.

This special stop is called the depot, and acts much like a gas station. A bus will divert its route and travel to the depot when it is running low on fuel. The client has stated that you do not have to worry about fuel management – you may safely assume that your buses always have sufficient fuel to make it to their next stop. Also, the client will provide more clarification on how they would like the simulation to handle passengers as opposed to the way passengers are handled now.

Simulation Management

The system can be started using the following command:

```
java –jar marta_sim_13in_verbose.jar test_scenario.txt
```

The **marta_sim_13in_verbose** program reads the scenario data from the **test_scenario** text file, and then renders the information on the display screen ideally sized for a laptop. If you have a larger desktop screen, feel free to use **marta_sim_23in_verbose.jar** instead. The system then allows the simulation user to click the "**Move Next Bus**" button, which selects the bus with the lowest logical time, and transfers that bus to its next stop based on its assigned route. That bus also has its passenger count updated based on the passenger arrivals and departures at the new stop, which are determined by probability distributions. The "**Reset Buses**" button returns the state of the simulation back to the original setup.

Simulation Monitoring

Note that the buses display information that helps you monitor the state of the simulation. Each bus displays an information line in the following format:

```
b:<bus-id>->s:<stop-id>@<time>//p:<riders onboard>/f:<distance remaining>
```

For example, suppose there's a bus at stop #8 (West End) with the following information line:

```
b:63->s:9@16//p:1/f:95
```

Then this means that bus #63 is traveling to stop #9 (Sports Stadium), and will arrive at logical time 16. The bus is currently carrying 1 rider, and has enough fuel to travel 95 more miles. This information is good for a start, but you will be asked to store more information about the state of the information, and to perform more analysis about the simulation in order to determine how effective the current setup of buses, stops and routes are supporting the bus passengers.

Calculating Distances and Travel Times

We use relatively simple methods to calculate the distances and travel times for buses moving between bus stops in your system. First, the locations of the bus stops are given in terms of geographical latitude and longitude, which is normally measured in units using (Degrees / Minutes / Seconds) notation. We will calculate the distance in (statute miles) for our purposes. To do this, we will first calculate the distance between bus stops using a simple Euclidean distance formula.

We understand that there are other factors that could affect the true distance:
(1) The Earth is curved: agreed, but we will be calculating relatively (very) short distances, so we will accept the minor errors that come with using a "flat surface" approach; and,

(2) The buses travel along roads which might be curved: also true, but we will also accept these minor errors in the same vein as using basic calculus techniques - we approximate the distance of a curved line by using shorter, straight line segments. We will make the reasonable assumption that there is some relatively straight path (e.g. sequence of roads) between the two stops so any road curvature is insignificant.

You can think of the longitude and latitude for a bus stop as X and Y coordinates, respectively. If bus stop #1 is located at (x1, y1), and bus stop #2 is located at (x2, y2), then the distance in Degrees/Minutes/Seconds (or [d/m/s]) will be approximated as:

- distance [d/m/s] = square_root $((x1 - x2)^2 + (y1 - y2)^2)$

Using data from real MARTA bus stop locations, we also determined that using a conversion factor of 70 will produce approximate answers in units of Statute Miles (or [miles]):

- true_distance [miles] = 70 * distance [d/m/s]

Finally, to calculate travel times, we will use a simple distance = rate * time relationship. We track the speed of a bus in terms of Miles per Hour (or [mph]). To make our simulation more manageable, we would like to track events at a more granular level than hours - we would prefer to use minutes. Consequently, we will also use a conversion factor to calculate the travel time of a bus from stop A to stop B as:

- travel_time [minutes] = 60 [minutes_per_hour] * distance [miles] / rate [mph]

The nice part of this is that it helps "standardize" the flow of time in our simulation system. For example, if an event occurs at "logical time" 7, and another event occurs at logical time 11, then the amount of "real time" between the two events would be approximately 4 minutes. This helps our clients keep some sense of real-world timing when using the system.

From a source code standpoint, the Java instructions needed should be relatively straightforward. For the distance conversions, code of this form should be sufficient:

```
double distance = 70.0 * Math.sqrt(Math.pow((stop1_latitude –
stop2_latitude), 2) + Math.pow((stop1_longitude – stop2_longitude), 2));
```

And for the travel time calculations, something of this form should suffice:

```
int travel_time = 1 + (distance.intValue() * 60 / bus.getSpeed());
```

Please note that the travel time calculations shown here also "convert" our distance calculations into an integer format. For our discrete-event simulation, we would prefer to have events occur on integer (minute) boundaries, so we can truncate answers as needed. We convert the possibly

fractional distance answer into an integer format (losing a bit of accuracy) before getting a travel time in (integer/non-fractional) minutes.

Also, note that the travel time will be at least one minute, even if the distance between stops is practically zero. This is another key aspect of our simulation system: events that take "zero time" can sometimes cause "infinite loops" in some cases. We often generate the time for new events based on the time of the current event plus the travel time; and, when travel time is zero, then the "new" event (effectively the same as the current event) gets put back into the head of our priority queue, causing unexpected cycles. This mandatory "travel time is at least one or more minutes" restriction is a safety factor to avoid these problems.

Evaluating Your UML Submissions

- You must generate your diagrams using an automated tool (e.g. Argo UML, LucidCharts, Microsoft Visio, etc.) so that they are as clear & legible as possible. Even PowerPoint is allowed, though this is an excellent opportunity to use a tool that is more appropriately designed for UML as opposed to a general drawing tool like PowerPoint. The choice of tool(s) is yours; however, you should do a "sanity check" to make sure that your final diagrams are readable when exported to PDF; or, if necessary, some reasonable graphical format (PNG, JPG or GIF).
- We prefer that you submit your diagrams in dark, rich colors (particularly black). Diagrams submitted in certain colors, especially pastel or lighter shades, can be difficult to read.
- You must designate which version of UML you will be using – either 1.4 (the latest ISO-accepted version) or 2.0 (the latest OMG-accepted version). We highly recommend that you label all diagrams with a header that includes your name and the UML version being used. There are significant differences between the UML versions, so your diagram must be consistent with the standard you've designated. Either version is acceptable at this point in the course.
- You are permitted to add a few sentences to explain any aspects of your design that you feel need extra clarification. These sentences are optional: non-submissions will not be penalized.

Writing Style Guidelines

The style guidelines can be found on the course Udacity site, and at:
https://s3.amazonaws.com/content.udacity-data.com/courses/gt-cs6310/assignments/writing.html

Closing Comments & Suggestions

This is the information that has been provided by the customer so far. We (the OMSCS 6310 Team) will likely conduct an Office Hours where you will be permitted to ask us questions in order to clarify the client's intent, etc. We will answer some of the questions, but we will not necessarily answer all of them. Also, though this current version of the system is "the core" of the system going forward, our clients will very likely add, update, and possibly remove some of the requirements over the span of the course. One of your main tasks will be to ensure that your architectural documents and related artifacts remain consistent with the problem requirements – and with your system implementations – over time.

Quick Reminder on Collaborating with Others

Please use Piazza for your questions and/or comments, and post publicly whenever it is appropriate. If your questions or comments contain information that specifically provides an answer for some part of the assignment, then please make your post private first, and we (the OMSCS 6310 Team) will review it and decide if it is suitable to be shared with the larger class. Best of luck on to you this assignment, and please contact us if you have questions or concerns.