SURTS Business Layer Module Design Document

Version 3.0

Prepared by:
Nathan Bourgeois
Jonathan Larsen
David Ma
Thomas Pfeifer

12/14/2020

Document Revision History

Date	Version	Description	Author
12/02/2020	2.0	All initial design documents combined	Nathan, Jonathan, David, Thomas
12/14/2020	3.0	Added/Modified Interaction Diagrams and Class Diagram	Nathan, Jonathan, David, Thomas

Table of Contents

1. Introduction

1.1 Purpose

The purpose of this document is to describe a possible design for the business layer module of the SURTS system. As a preliminary design, it will not include details that are considered too low-level. The document is split into sections, laid out in the table of contents. The introduction will introduce the document and the system. The design overview will highlight the approach which we will use to build the system. The interfaces and data stores section will explain what already built components the system will interface with. The structural design section will show and explain the class diagram for the system. The dynamic model will show how classes interact with a sequence diagram. Finally, the last two sections will explain non-functional requirements and supplementary documentation briefly.

1.2 System Overview

In general, the SURTS system is meant to be a simulation of a transportation system of buses, routes, and passengers. The SURTS business layer module is a layer of the SURTS architecture that processes requests from the GUI and makes use of the data persistence layer in responding to those requests. Given the scope of the system, the design is purposely kept somewhat generic and does not include all the details included in a design for a more specialized system. This design document outlines an object-oriented design of the SURTS Business Layer Module. As iterated in the requirements document, the target users are any who might be curious about the organization of a transportation system e.g. statisticians, economists, transportation administrators, etc.

1.3 Design Objectives

The main role of the business layer module is to run the simulation itself with its various functionalities. This means that it will *not* be directly configurable. Configurations will be passed in from the GUI level. All of the heavy lifting of the simulation will take place in the BLM, after it is configured. This means managing the various aspects of the simulation such as routes, passengers, and buses , and making sure that those components are each simulated accurately. Specifically, this means that simulated buses will carry passengers to and from their destinations along a number of predefined routes.

The BLM also does not deal with the storage of logs and reports. Since the data needs to last beyond a single simulation run, it will be dealt with by the data persistence level, and the BLM will only need to make use of the interface provided by the data persistence level.

One assumption being made is that the GUI takes care of all access issues, which means that all requests made to the BLM are valid. This means that non-functional requirements such as business rules do not pertain to the BLM. This does not address performance.

The design objectives of this document are to provide enough information for a software engineer to implement the system and to provide enough detail to develop comprehensive tests for the system before the actual code is available and finished.

1.4 References

The Software Requirements Specification for SURTS available from the Assignment 3 submission.

1.5 Definitions, Acronyms, and Abbreviations

BLM:	Business Layer Module
SURTS:	System for Uniform Route-based Transportation Simulation
GUI:	Graphical User Interface
OO:	Object Oriented

2. Design Overview

2.1 Introduction

For the BLM, we choose to adopt an object-oriented design. This is advantageous for many reasons, but was ultimately chosen for the simplicity of handling requests. The proposed architecture of the system is a 3-layered one. The user interacts with the GUI, which makes requests to the BLM, which makes requests to the data persistence layer. This allows for the ability to easily make changes and the use of adapters when new or modified functionality is required.

2.2 Environment Overview

Since the requested environment was not specified in the requirements, this is going to be left up to the implementation team for our preliminary design.

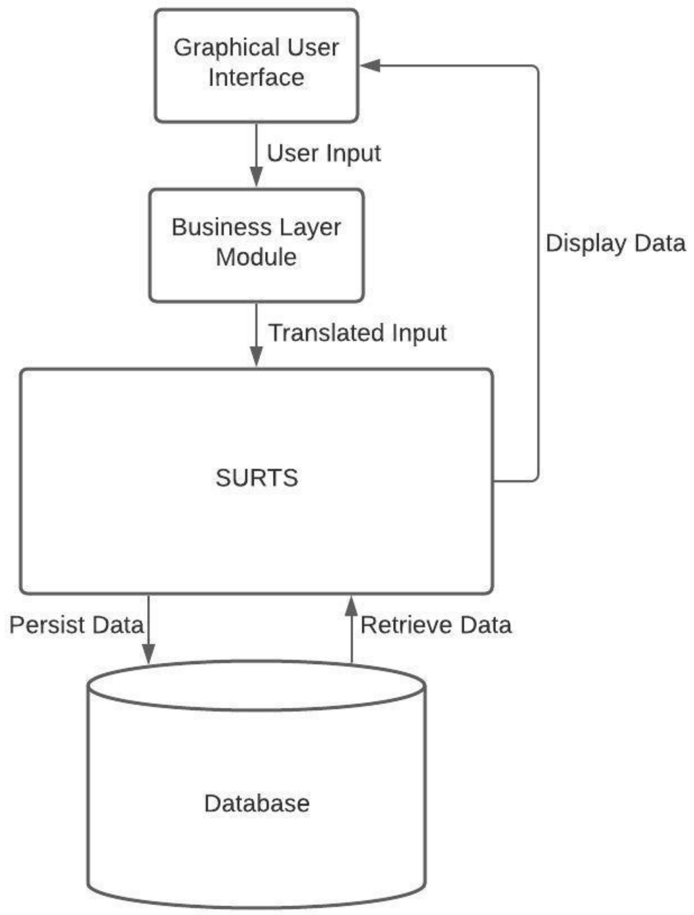
2.3 Constraints and Assumptions

As described in the requirements document:

- "The system will have a user interface. This interface will allow the user to modify simulation parameters, run the simulation, view simulation logs, and generate reports"
- "The system will be able to run on a device running some kind of operating system, where the device has sufficient computational power and speed to run the system. The system will not need to support mobile devices"
- Before a simulation may be run the default values for the system must be configured
- All communication shall be conveyed through the user interface.

2.4 System Architecture

The architecture of the system is relatively basic. The software will receive command through a GUI which will be translated into actionable commands to the system via the BLM. The system will also be able to store its data to the database layer for later retrieve back to the SURTS system, essentially resuming the system where it left off.



*SURTS is more thoroughly described in sections: 3.2, 3.3, 3.4, 4

2.5 System Interfaces

The interfaces the system interacts with are the BLM for translating user commands from the GUI and the database for storing and retrieving data. The interfaces to the GUI and database are not known at this time and must be taken into account in the design and implementation of the SURTS system.

2.6 Business Layer Module

This interface will send the client requests in a usable form down to the SURTS system to be carried out there. This has already been defined for us. We need only to be able to handle all the types of requests.

2.7 Data Layer

The data layer will allow for the persistence of objects from the system. This has already been implemented for us. This layer has implemented two functions:

Persists(Object obj) will persist the state of an object

Retrieve(UUID ident) will retrieve the object from the data layer

3. Design

3.1 Introduction

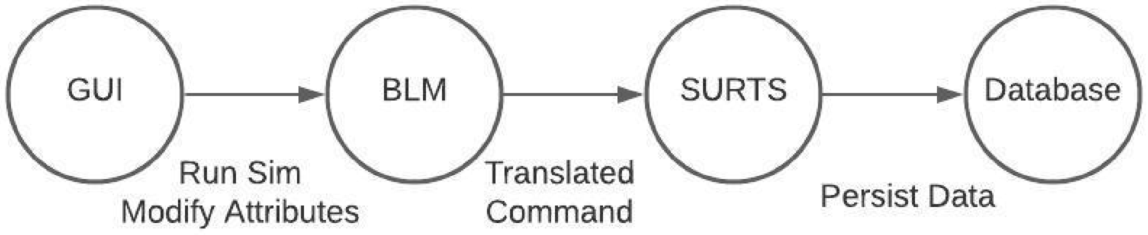
The design is split into the two sections of functional decomposition of the system and the design of the abstract data types on which the system operates.

3.2 Data Flow Diagram

At the context level, the SURTS system will receive translated commands from the GUI via the BLM. There are the two following outcomes based on the user input.

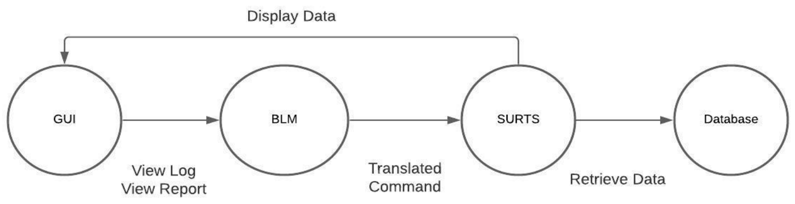
3.2.1 Storing data

The system will store modified attributes or log information in the database for future use and access.



3.2.2 Retrieving data

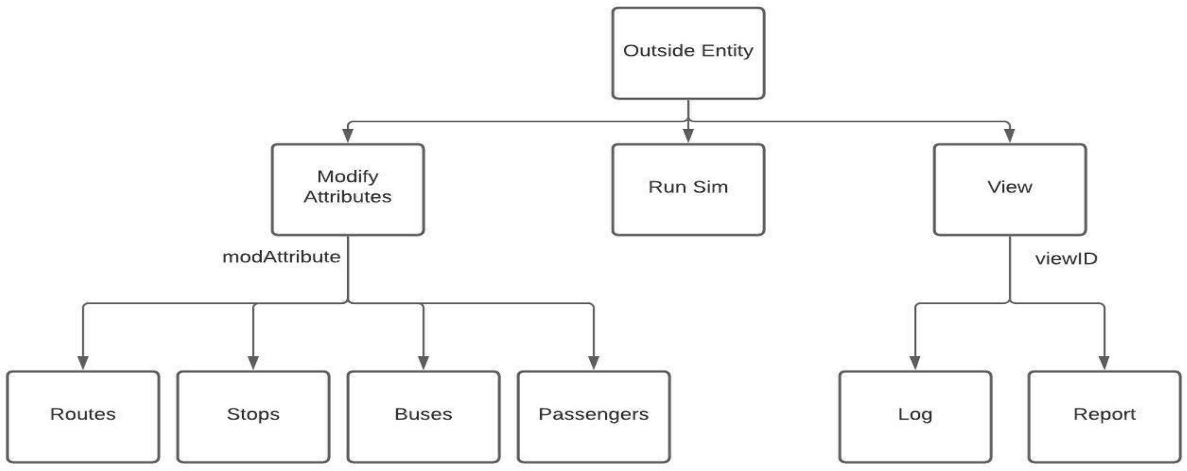
The system will display log or report information from the database to the user through the GUI.



3.3 Functional Decomposition

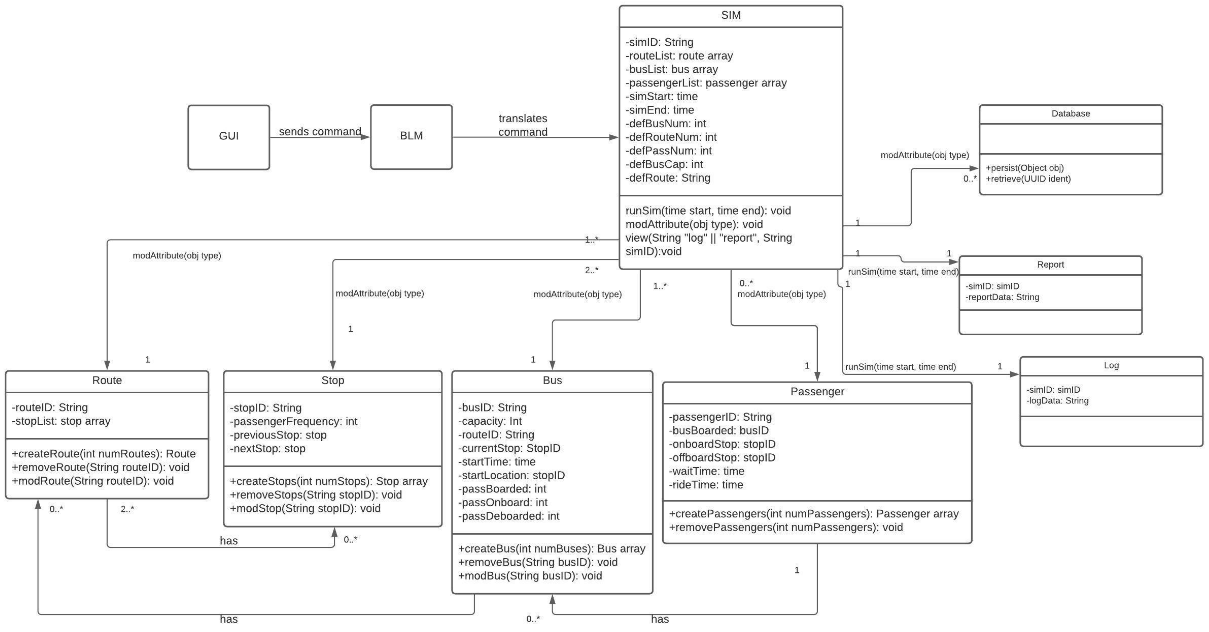
The program will have all processing and function calls governed by outside entities. The function will receive user commands from the GUI, which will be translated in the BLM. It will then invoke the appropriate functions from the appropriate classes as shown below.

*A more detailed definition of each function is given in section 4.1



3.4 Abstract Data Types

The data types needed to run and track information about the simulation are designed using OO techniques and UML. The following is a class diagram.



The SIM class has three methods. The runSim method runs the simulation using the defined route, bus, and passenger lists or their respective default values. The view method is used to view either a log or report of a simulation with the entered simID or over the specified time frame. The information to display these is retrieved from the database. Finally the modAttribute method takes an object type as an argument and has the respective object type call either create, remove, or mod methods to modify the appropriate list in SIM. All variables are private and all methods are public. The set and get methods for setting and getting variables have been omitted for the sake of a more readable diagram.

4. Detailed Function and Class Definitions

4.1 Function Definitions

4.1.1 Function runSim

Signature: void runSim(time start, time end)

Parameters: start time: time signature to begin simulation
end time: time signature to end simulation

Result: runs simulation, stores log information, displays report

Processing: runSim uses either the default values or user defined values of routes, buses, and passengers to run the transport simulation from start to end. Upon finishing, run sim stores log information in the database and displays a report to the user interface.

4.1.2 Function modAttribute

Signature: void modAttribute(obj type)

Parameters: obj type: the type of what the user wants to modify; either stop, bus, route or passenger.

Result: modAttribute selects the object of the correct type to be modified.

Processing: modAttribute takes in the type of object to be modified. With this object it calls either create, remove, or mod on the respective object type and updates the list within the SIM class.

4.1.3 Function view

Signature: void view(String type, String simID)

Parameters: String type: either "log" or "report"

String simID: the ID of the sim whose log or report is to be viewed

Result: the log or report of the sim with the matching ID is displayed to the user interface.

Processing: view calls retrieve(5.1.12) and locates the log stored in the database with the matching simID. Then either the log is displayed, a report is displayed generated from that log, or an error message is displayed in the event that no log exists.

4.1.4 Function modStop

Signature: void modStop(String stopID)

Parameters: String stopID: the stop to be modified

Result: The stop with the matching stopID is modified.

Processing: The stop ID or passenger frequency can be modified via the interface.

4.1.5 Function modRoute

Signature: void modRoute(String routeID)

Parameters: String routeID: the route to be modified

Result: the routes stops are modified

Processing: the route with matching routeID allows the user to call createStops, removeStops, or modStop on each stop in the stopList and the routeID can be modified via the interface.

4.1.6 Function modBus

Signature: void mosBus(String busID)

Parameters: String busID: the bus to be modified

Result: the bus with the matching busID is modified.

Processing: The busID, capacity, routeID, startTime, and startLocation can be modified via the interface.

4.1.7 Function createStops, createRoutes, createBuses

*the following three methods function the same and are combined here for ease of understanding

*_ represents stop, route, and bus respectively

Signature: _array create_(int num_)

Parameters: int num_: the number of stops, routes, or buses to create.

Result: The entered number of stops, routes, or buses are created and added to their respective lists in the SIM class.

Processing: The number entered, objects are created and allow their variables to be set by the user via the interface.

The following are the variables set for each object.

Stop: stopID, passengerFrequency

Route: routeID, stopList

Bus: busID, capacity, routeID, startTime, startLocation

4.1.8 Function createPassengers

Signature: passenger array createPassengers(int numPassengers)

Parameters: int numPassengers: the number of passengers to be created

Result: A number of passengers are created and added to the passengerList in the SIM class.

Processing: A number of passengers are created. Their onboard and offboard locations are randomly selected based on the available routes and stopFrequency. Their wait and ride times are set to 0.

4.1.9 Function removeStops, removeRoutes, removeBuses

*the following three methods function the same and are combined here for ease of understanding

Signature: void remove_(_ID)

Parameters: _ID: the id of the stop, route, or bus to be removed

Result: the stop, route, or bus with matching ID is removed from their respective list in the SIM class.

Processing: The object with the matching ID is deleted from their respective list in the SIM class.

4.1.10 Function removePassengers

Signature: void removePassengers(int numPassengers)

Parameters: int numPassengers: the number of passengers to be removed from the list in the SIM class.

Result: a number of passengers are removed from the passenger list in the SIM class.

Processing: a number of passengers are removed at random from the passenger list in the SIM class.

4.1.11 Function persist

Signature: void persist(Object obj)

Parameters: obj: the object that is to be maintained in the database

Result: the object passed in is maintained in the database for future queries or simulations

Processing: the objects state is stored in the database. For attributes, their values are saved for use in future simulations and for logs or reports their data is kept for the purpose of viewing.

****when objects are modified, their state is persisted in the database. This means there is an implicit call to persist after an object has been modified.***

4.1.12 Function retrieve

Signature: Object retrieve(UUID ident)

Parameters: ident: the unique ID of what object is to be retrieved

Result: the object with the matching UUID is returned for use in a report, log, or simulation

Processing: the object is returned to the caller. For attributes, their values are known for the next simulation and for logs or reports their data is displayed to the GUI.

4.2 Class Definitions

4.2.1 SIM

Overview: This class stores default values along with lists of other class objects. This class serves as the data store which is used to maintain persistent data.

Attribute Description:
Attribute: simID
Type: String
Description: The unique ID of the simulation
Constraints: cannot be empty, is generated up runSim start

Attribute: routeList
Type: route array
Description: a list containing all route objects to be used in the simulation
Constraints: must contain default route

Attribute: busList

Type: bus array
Description: a list containing all bus objects to be used in the simulation
Constraints: must contain default bus

Attribute: passengerList
Type: passenger array
Description: a list containing all passenger objects to be used in the simulation
Constraints: none

Attribute: simStart
Type: time
Description: the start time of the simulation
Constraints: must be time before simEnd

Attribute: simEnd
Type: time
Description: the end time of the simulation
Constraints: must be time after simEnd

Attribute: defBusNum
Type: int
Description: default number of buses
Constraints: none

Attribute: defRouteNum
Type: int
Description: default number of routes
Constraints: none

Attribute: defPassNum
Type: int
Description: default number of passengers
Constraints: none

Attribute: defBusCap
Type: int
Description: default bus capacity
Constraints: none

Attribute: defRoute

Type: String
Description: default route ID
Constraints: none

*see 4.1 for method descriptions

4.2.2 Route

Overview: This class stores all information for using route.

Attribute Description:

Attribute: routeID
Type: String
Description: the unique ID of the route
Constraints: none

Attribute: stopList
Type: stop array
Description: a list of stop objects that are contained within the route
Constraints: must have at least 2 stops

*see 4.1 for method descriptions

4.2.3 Stop

Overview: This class contains the information needed to represent a bus stop.

Attribute Description:

Attribute: stopID
Type: String
Description: the unique ID of the stop
Constraints: none

Attribute: passengerFrequency:
Type: int
Description: a number to determine what percentage of passengers have this on/off board stop.
Constraints: the sum of all passengerFrequencies must be 100

Attribute: previousStop

Type: stop
Description: a reference to the stop object that is before this stop on a route
Constraints: this stop must be on the same route as previousStop

Attribute: nextStop
Type: stop
Description: a reference to the stop object that is after this stop on a route
Constraints: this stop must be on the same route as nextStop

*see 4.1 for method descriptions

4.2.4 Bus

Overview: This class contains all information needed to represent a bus.

Attribute Description:

Attribute: busID
Type: String
Description: the unique ID of the bus
Constraints: none

Attribute: capacity
Type: int
Description: the maximum capacity of the bus
Constraints: must be a positive number

Attribute: routeID
Type: String
Description: the route ID the bus will travel
Constraints: the route ID must be in the routeList

Attribute: startTime
Type: time
Description: what time in the simulation the bus will begin its route
Constraints: the start time must be equal to or later than the simStart

Attribute: startLocation
Type: String
Description: the stopID of where the bus route will begin
Constraints: the stopID must be in the stopList of the bus's route

Attribute: passBoarded
Type: int
Description: the number of passengers boarded
Constraints: none

Attribute: passOnboarded
Type: int
Description: the number of passengers on boarded
Constraints: none

Attribute: passDeboarded
Type: int
Description: the number of passengers off boarded
Constraints: none

Attribute: currentStop
Type: stop
Description: a reference to the stop where the bus is currently
Constraints: the stop must be on the route with the same routeID as this bus

*see 4.1 for method descriptions

4.2.5 Passenger

Overview: This class contains all the information needed to represent a passenger.

Attribute Description:

Attribute: passengerID
Type: String
Description: the unique ID of the passenger
Constraints: none

Attribute: busBoarded
Type: String
Description: the busID of the bus the passenger boarded
Constraints: none

Attribute: onboardStop
Type: String

Description: the stopID the passenger will get on the bus
Constraints: the onboardStop must be on a route in the SIM routeList

Attribute: offboardStop
Type: String
Description: the stopID the passenger will get off the bus
Constraints: the offboardStop must be on the same route as the onboardStop, and in the SIM routeList

Attribute: waitTime
Type: time
Description: the time the passenger waited to onboard a bus
Constraints: none

Attribute: rideTime
Type: time
Description: the time the passenger rode the bus
Constraints: none

*see 4.1 for method descriptions

4.2.6 Log

Overview: This class contains all information needed to generate a log.

Attribute Description:

Attribute: simID
Type: String
Description: the ID of the simulation run whose data is contained in this log
Constraints: the simID must be of a simulation that has previously run

Attribute: logData
Type: String
Description: this is a formatted string that contains all necessary data that is to be stored in a log
Constraints: none

4.2.7 Report

Overview: This class contains all information needed to generate a report.

Attribute Description:

Attribute: simID
Type: String
Description: the ID of the simulation run whose data is to be used to generate this report
Constraints: the simID must be of a simulation that has previously run

Attribute: reportData
Type: String
Description: this is a formatted string that contains all necessary data that is needed to generate a report.
Constraints: none

5: Dynamic Model

The purpose of this section is to model how the system responds to various events, i.e. model the system's behavior. We do this using UML sequence diagrams.

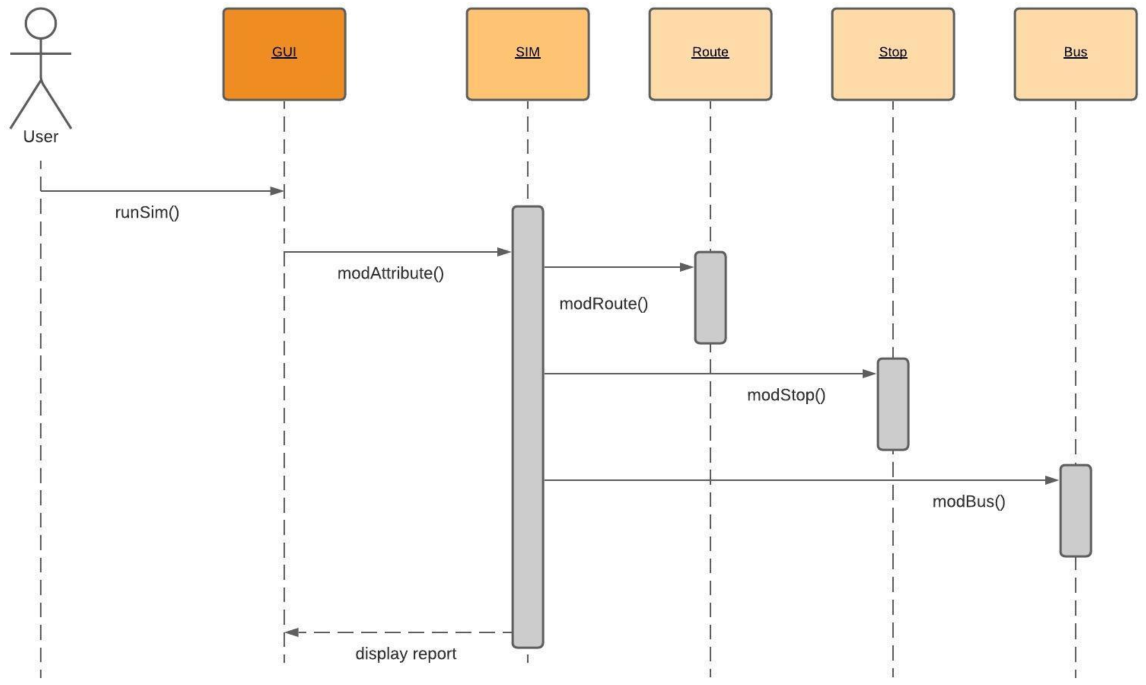
5.1 Scenarios

Each scenario has a subsection with the following information:

- Scenario Name: the name of the scenario
- Scenario Description: a brief description of what the scenario is about and the sequence of actions that take place
- Sequence Diagram: a sequence diagram showing various events and their relative time ordering

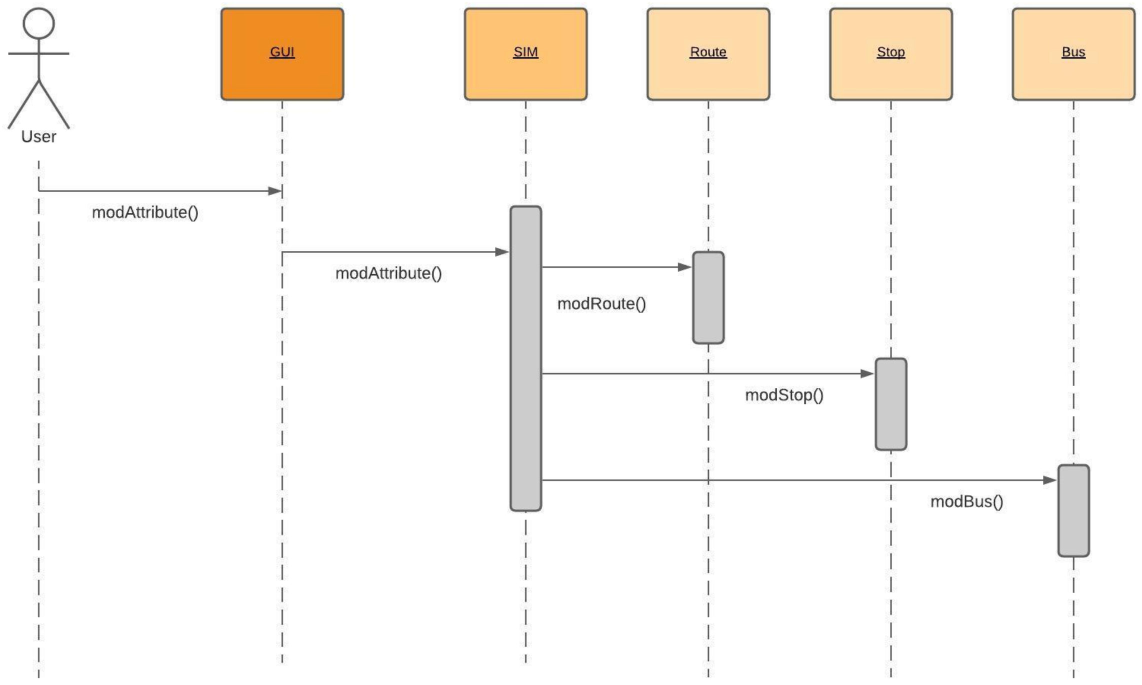
5.1.1 runSim()

This describes the main use case. The user runs the simulation with the default values or modifies one or more of the configurable attributes. The system runs to completion, displays a report for the simulation that has just run, then terminates.



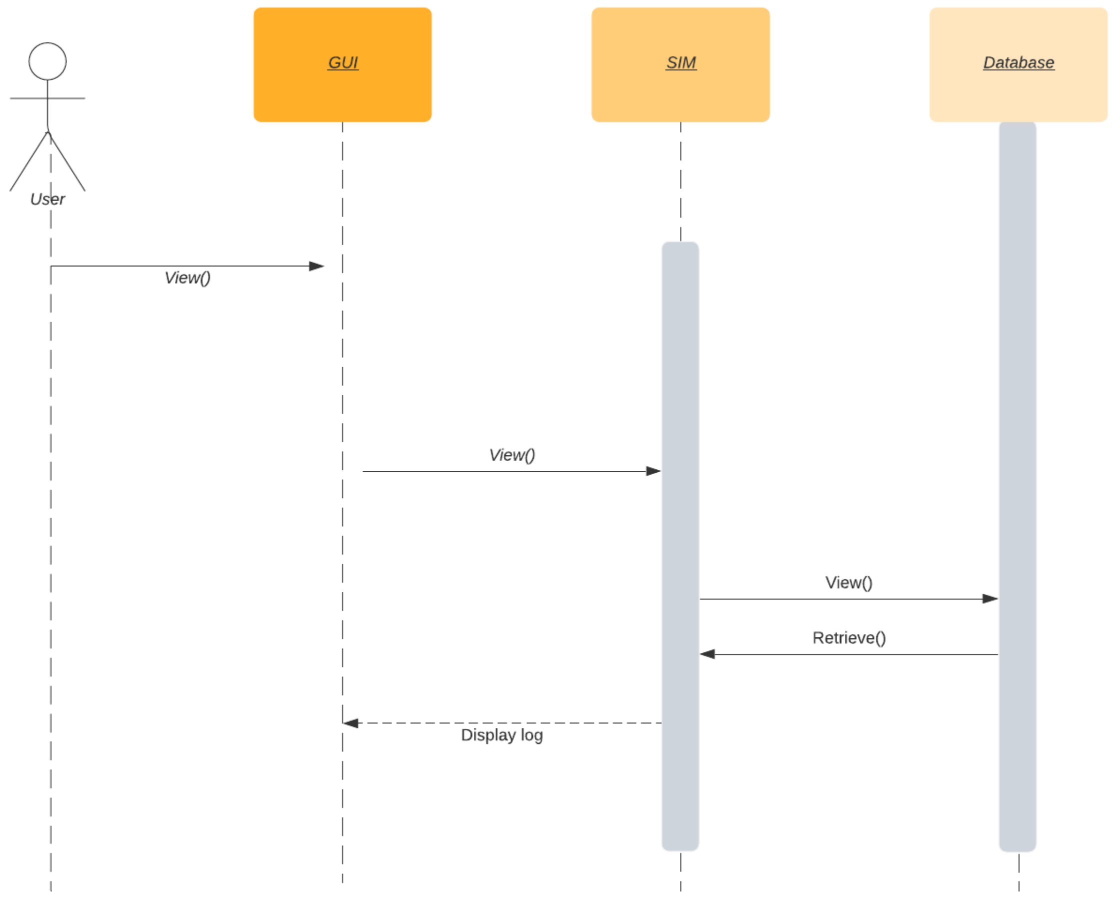
5.1.2 modAttribute()

The user wants to modify one or more of the configurable attributes in the system. The object in the SIM to be modified is changed to the configuration the user specifies.



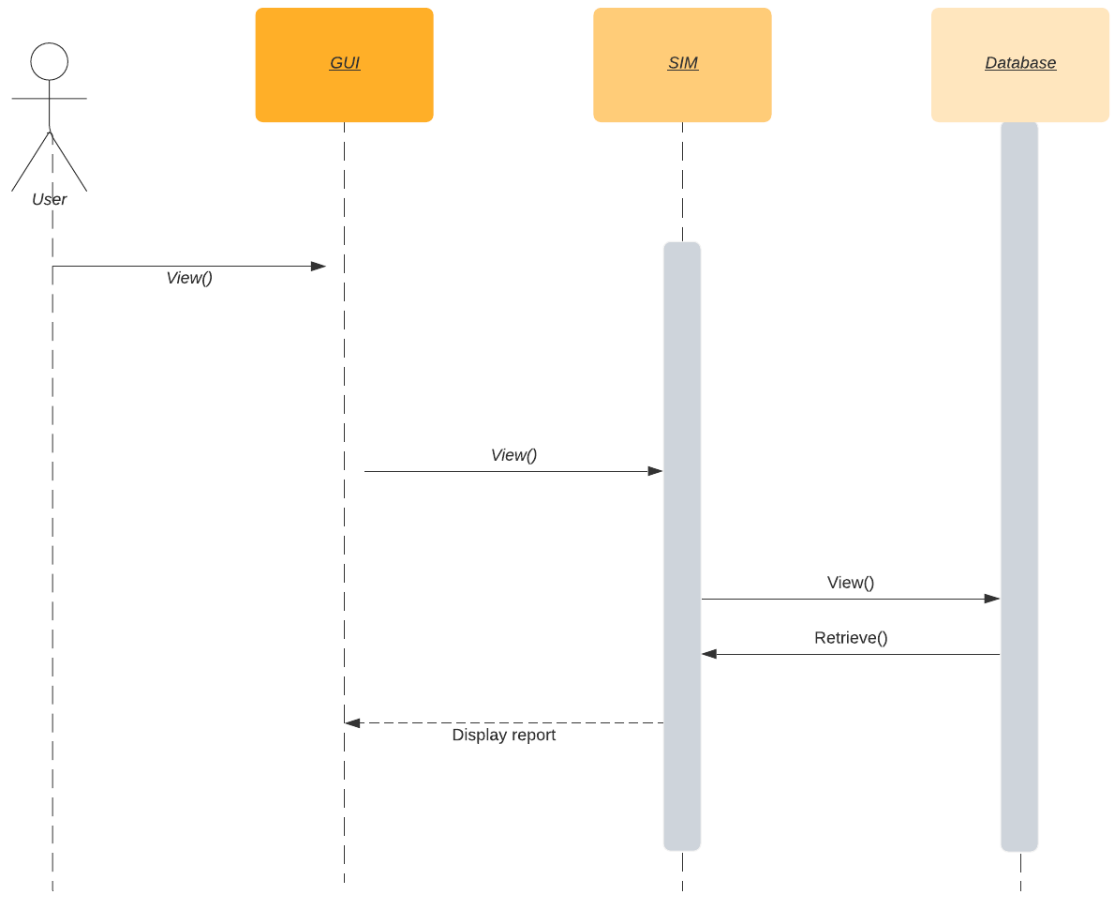
5.1.3a view()

The user wants to view a log or report from a simulation. The simulation accesses the database and retrieves the log. It is then displayed in the GUI.



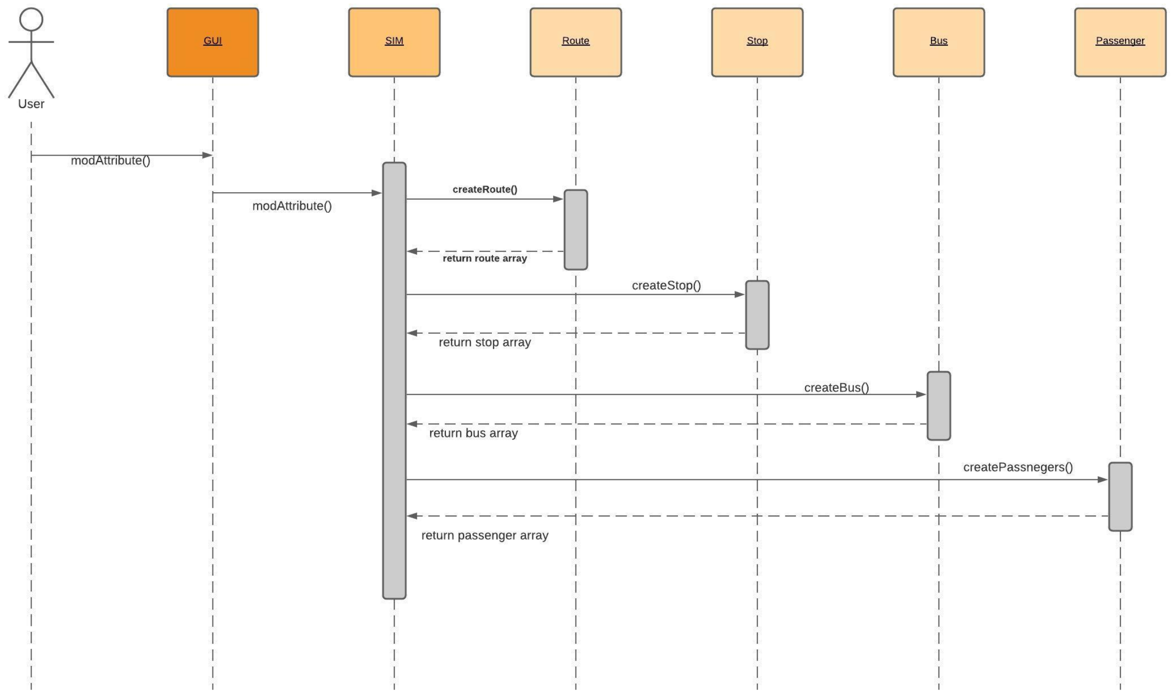
5.1.3b view()

The user wants to view a log or report from a simulation. The simulation accesses the database and retrieves the Report. It is then displayed in the GUI.



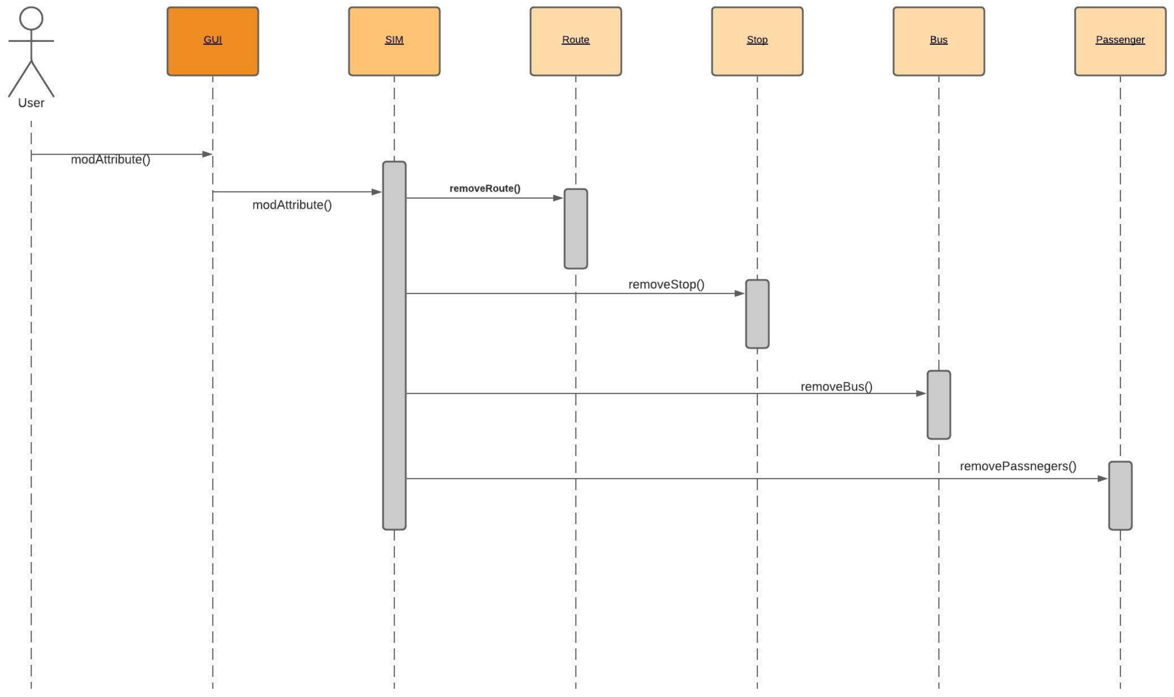
5.1.4 create()

The user wants to create a new route, stop, bus, or passenger. The respective object is created and stored in the SIM objects appropriate array for use in the next simulation.



5.1.5 remove()

The user wants to remove a route, stop, bus, or passenger from the SIM object. The respective object is removed from its array.



6: Design Rationale

The main rationale for most of the design choices in this document were to make the system as generic as possible. In terms of classes and functions, we tried to capture only the most essential components that needed to be stored. Other values that could be derived from these components have been left out, leaving room for more specific uses. For example, we did not include average ride time for passengers as it can be derived from the total number of passengers and their wait times. Excluding this derived value leaves the system open to interpretation and leads to the user thinking about what information they actually want to capture rather than assuming what they are trying to achieve by running the simulation. This also slightly reduces the size of each class in terms of stored data, which leads to more easily understood design.

Another design decision we made was to maintain simulation data in the SIM class object. This allows for slight alterations in the simulation parameters for repeated testing rather than having to reenter all new data to run a new simulation. The cost of storing one object is negligible in comparison to the time it would take to re-enter data for every simulation run.

We also included the use of default simulation values. This is most likely to be used at an administrator level but we feel that it increases the ease of use for the system. With the use of these default values for system parameters, we make sure that the user doesn't enter invalid data, such as having a number of passengers lower than 0. By using these default values, we are providing a safety net for users who may enter values incorrectly as well as providing a form of defense for more nefarious users. The user is signalled when a default value is going to be used so they have a chance to change it if they want and if they don't have a preference for a certain value such as what routes are in service, they don't have to enter anything.

An interesting design choice we also made was in terms of passengers. We decided that the user would only need to specify the number of passengers to be used in the simulation. They do not need to specify each of their individual boarding and off boarding locations. The reason for this decision was two fold. First, if we are to run the simulation for any extended amount of time, the number of passengers may very well be in the thousands. Entering two pieces of data for thousands of individual passengers would be very time consuming and overall very inefficient. Secondly, we decided to have the bus stops have a frequency. This means that each stop has a frequency of passengers that arrive per time interval that can be specified by the user. This fixes our first problem of mundane data entry and leads to what we feel is a benefit for the system as a whole. Using frequencies, the passengers' on and off boarding locations are generated somewhat more randomly. They may be 90% likely to choose stop one, but there is still a chance that they choose another. This better reflects how a real world transport system would actually function.

Finally, we make use of a facade in our design. This makes the modification of the actual code more accessible. Rather than having to change interactions between the GUI and the database every time some functionality is added or changed, with the use of an adapter in the BLM, we are able to translate GUI commands so that they function as intended in the SURTS system. By making this choice, the system as a whole will be more easily maintained while being open to extension, potentially increasing the longevity of the system and allowing for future modification if the need arises.

7. Non-functional Requirements

We must meet performance constraints on the system.To make sure that our simulation is satisfactory, we will instantiate the system with a normal load, and the simulation run time will be checked to see if it is adequate. Then the simulation will be run with an excessive load to see if it is handled appropriately. Finally, the system security and safety requirements will be met by implementing the system using a programming language with a support team behind it. This will also meet the business requirements of the system.

8. Supplementary Documentation

Refer to the A36 group submission under Nathan Bourgeois' name for more documentation components that may be helpful if needed.