Software Design Description

for

North Shore Extension for Port Authority of Allegheny County

Version 1.0 approved

Prepared by Hash Slinging Slashers

COE 1186, Fall 2017

1 November 2017

VERSION HISTORY

Any necessary revisions to the Software Design Descriptions (SDD) must be raised to an authorized Configuration Control Board (CCB) SDD Curator, which at the time of this writing exclusively includes Kevin Gilboy and Michael Kotcher. The revision must be overseen by one of the aforementioned SDD CCB members, then undergo subsequent approval by a separate CCB member within a week's time of the revision.

Versions shall start with 1.0, increasing by 0.1 for minor revisions and 1.0 for major revisions. Revision magnitude will be assessed at the time of the revision by the revisor.

Version Number	Implemented By	Revision Date	Approved By	Approval Date	Description of Change
0.0	Kevin Gilboy	09/13/17	Michael Kotcher	09/13/17	Creation of template
0.1	Michael Kotcher	09/13/17	Kevin Gilboy	09/13/17	Addition of Section 3 template
1.0	The Team	10/31/17	Kevin Gilboy	11/1/17	Creation of document

TABLE OF CONTENTS

1 OVERVIEW

1.1 SCOPE

This software design description (SDD) is written by the North Shore Extension software development team and intended for both the developers and stakeholders of the project. It is meant to explicitly convey the overall software architecture in a way that 1) allows the stakeholders to verify that requirements are being met, 2) allows the stakeholders to retain documentation for future maintenance and internal development, 3) conveys a stable design blueprint for developers to begin feature creation.

1.2 PURPOSE

This SDD document is intended to express the software architecture for the North Shore Extension from multiple vantage points and levels of granularity.

1.3 INTENDED AUDIENCE

This SDD document is intended for technical and non-technical backgrounds alike. It provides a UML-based graphical overview of module use cases, sequences, and classes pertinent to the overall system function without delving into code syntax or implementation details.

1.4 CONFORMANCE

The conformance to the images and diagrams put forth in this SDD can be broken down as follows:

- GUIs - The GUIs reflected in this work are meant as a potential implementation of the functionality that each module presents, and there is a possibility that they may not reflect the final user interfaces. They are not meant to be binding in any way. The GUIs in this document will be kept up to date with those in production as the project evolves.
- Use Case Diagrams - The overall functionality represented in the use case diagrams accurately reflects the functional requirements that will be included in this project.
- Sequence Diagrams - The overall functionality and sequence of the sequence diagrams in this SDD are accurate. However, as the result of high-level abridgement, the functional names of the data flows may be fictional to better represent the function they are performing. In other words, sequence flows may have different names than the functions that they call in the implemented code as a way to be more representative of their function. This is essential for clarity and brevity in this document; for example a flow may be called “getTrain(Name)” but may use several methods that: detects a user’s click, determines the train that the user clicked on, fetches the train object, and gets the train name. In addition, the list of sequence diagrams in this SDD may not be all inclusive, as this would be unreasonably long. The use cases selected for sequence diagram rendering were the ones decided to be most pertinent and involved. Additional, reasonable sequence diagrams may be provided at request by contacting the

chair of the CCB, who at the time of writing is Kevin Gilboy and may be reached at kevingilboy@pitt.edu.

- Data Design - All data structures outlined in this SDD are accurate to code implementation, however not all-inclusive. Several, uninvolving data aggregations have been omitted for brevity.
- Class Diagrams - All class diagrams in this SDD are accurate and all-inclusive to the functions that are/will be implemented in code, and will be kept updated on an ongoing basis.

2 DEFINITIONS AND ACRONYMS

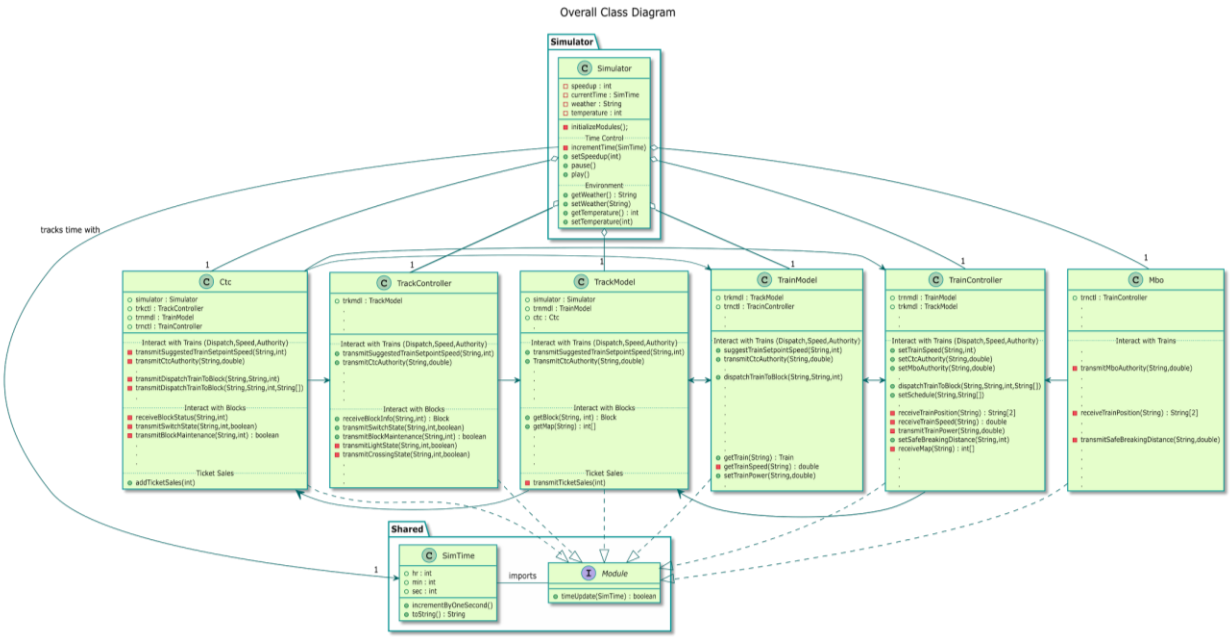
- **SDD** - Software design description. This document is an SDD which contains architecture and sequence diagrams for the various components that comprise the North Shore Extension.
- **STP** - Software test plan. These documents have been provided along with this SDD to describe the planned tests and deadlines necessary for this project.
- **GUI** - Graphical user interface. It is essentially what a user uses to interact with the system.
- **Overall** - Something that pertains to the entire system.
- **Use case diagram** - A diagram outlining the various actors and their potential interactions with each module. Use cases in this SDD are defined as “front-end” interactions, so the subtle “behind-the-scenes” transmission of data from module to module is not a use case by this definition.
- **Class diagram** - A diagram outlining the various classes along with their internal class variables and methods necessary for a functional unit, which in this case a module.
- **Sequence diagram** - A diagram used to show the flow of information between modules and their inner classes.

3 OVERALL ARCHITECTURE

Each individual North Shore Extension system module shall be encapsulated within a central Simulator class, which will be responsible for instantiating each module, establishing legal lines of communication, and hosting variables that are mandatory to the simulation but do not necessarily belong in any one particular module (time, simulation speed, weather). Such implementation of an overarching, omniscient class as an “overall repository” is common practice for synchronizing many subsystem modules together.

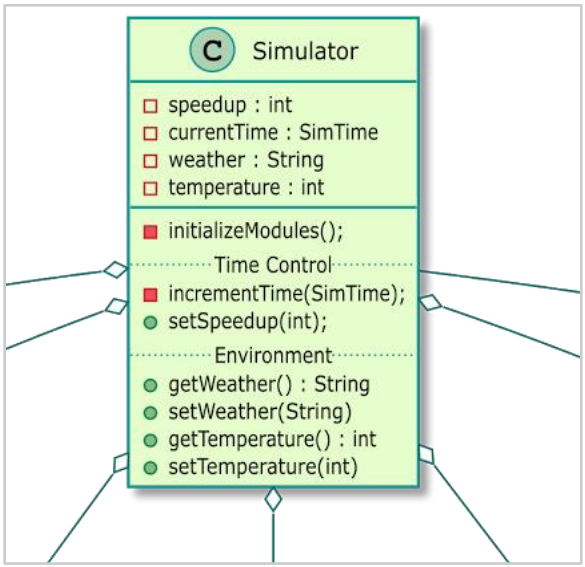
The class and state diagrams below describe the implementation of the Simulator class in great detail.

3.1 CLASS DIAGRAM



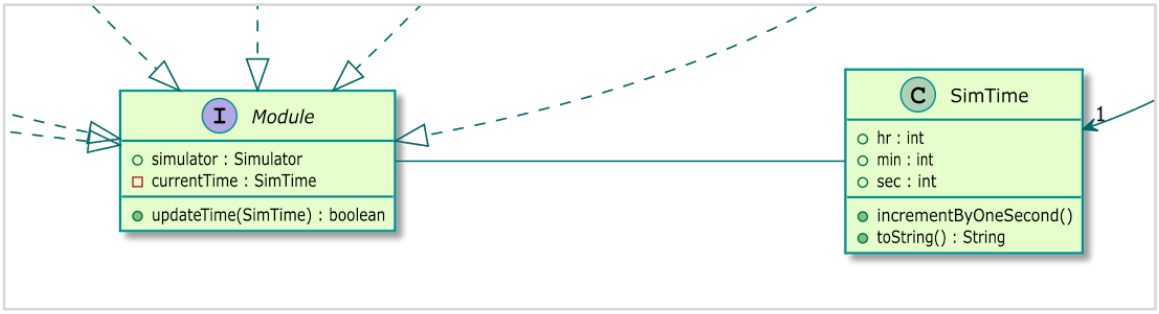
An overall view of the Simulator class is shown above, with zoomed-in views shown below for enhanced visibility.

Simulator Cass for Submodule Initialization and Assignment of References



System-Level Module Relationships

Module and SimTime Classes for Synchronization

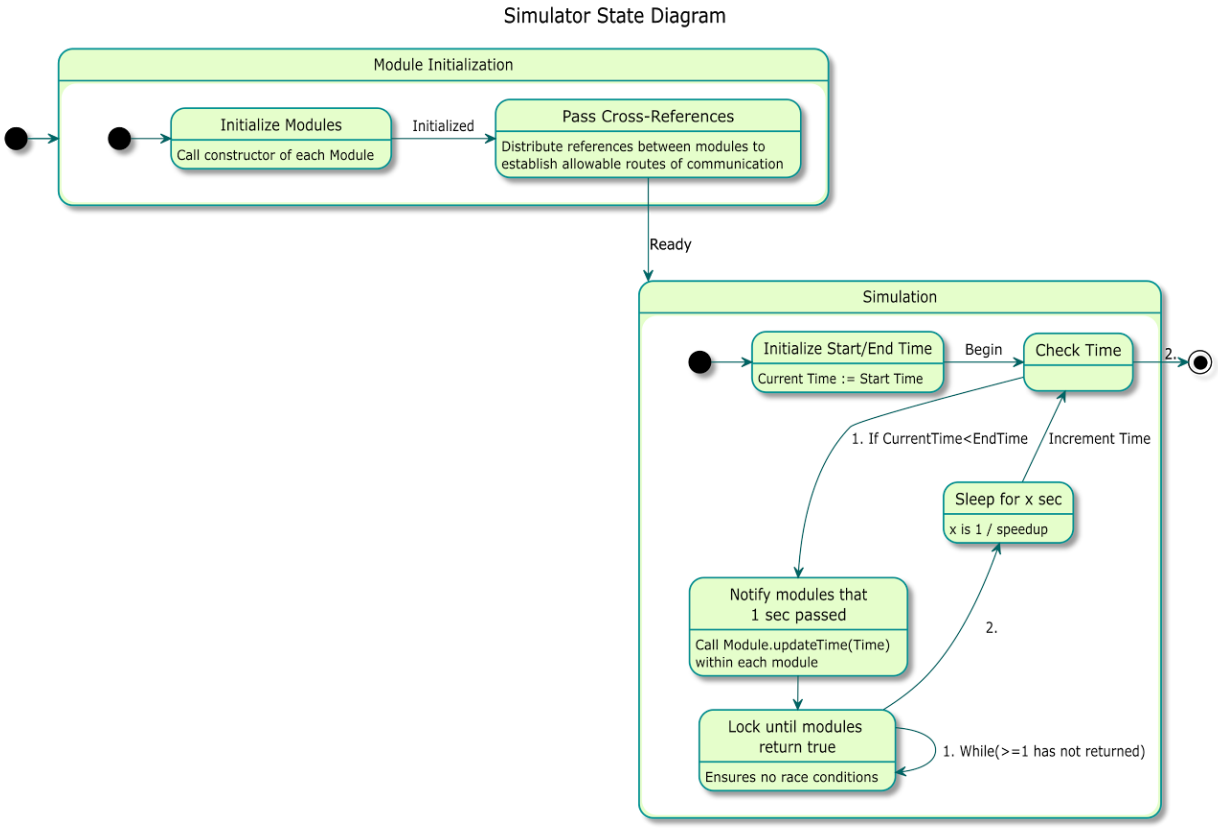


The Simulator class initializes each of the modules in the system, and distributes references between modules as depicted by the connections between the six modules shown above. This is meant to allow uni- or bi-directional communication between appropriate modules.

The Simulator class also contains environmental variables, which are physically realizable, ambient characteristics of the environment. Such variables are weather, temperature, and relative simulation speed. The CTC will be given the ability to change these factors as a means of simulation.

In addition, a Module interface containing the updateTime(SimTime) method is implemented by each sub-system to maintain synchronization with the overall system clock. The SimTime class, which is imported by Module, is a custom time object that represents a relative time within the simulation. The calling of updateTime(SimTime) is used to pass the current simulation time to each submodule so that it may perform time-critical operations at every virtual passing of one second. A more in-depth description of what occurs during the simulation loop can be found in Section 3.2 State Machine.

3.2 STATE MACHINE



A state diagram for the Simulator's execution is shown above. Upon starting, the Simulator creates instances of the Ctc, TrackController, TrackModel, TrainModel, TrainController, and Mbo classes. Once initialized, the Simulator takes care of distributing legal references between modules that require a path of communication. At this point, the Simulator has created all the objects needed, and must begin playing the role of "time-keeper."

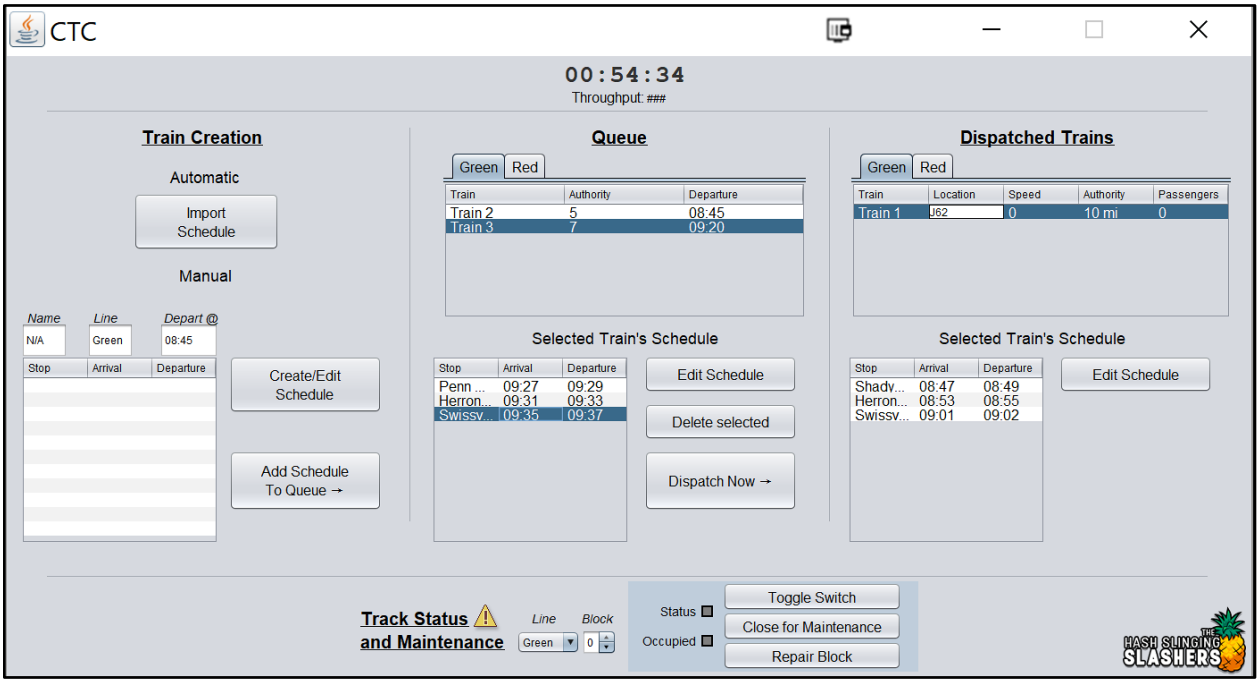
The simulator creates three new SimTime objects, one representing the start time, one representing the current time, and one representing the end time. The current time is initially set to the start time. At this point, the simulation loop begins, similar to how a clock runs the processor of a computer. If the current time is less than the end time, we call *updateTime(currentTime)* within every module. As shown in the class diagram in Section 3.1, every module implements the Module interface and is therefore contractually obligated to have an *updateTime(SimTime)* method. The modules are expected to handle this event as a "rising clock edge" letting them know that one virtual second has passed. The Simulator locks while

waiting for every module to return from *updateTime(currentTime)*. The Simulator then busy-waits for *1/speedup* wall-clock seconds before incrementing the virtual time by one second and looping. The simulation ends when the end time has been reached.

4 SYSTEM ARCHITECTURE BY MODULE

4.1 CTC

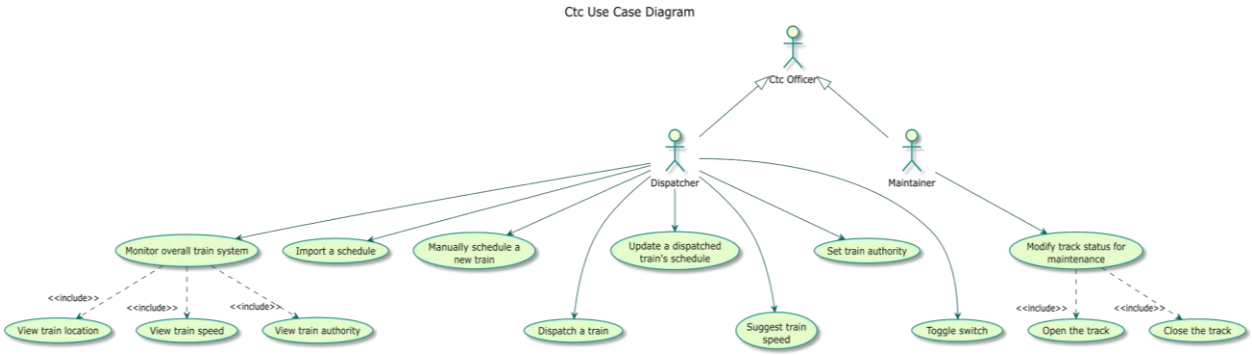
4.1.1 INTERFACE



The above GUI serves as a reminder of the CTC user interface. The GUI is navigated from left to right, representing the logical flow of creating, dispatching, and monitoring trains. The left-most pane allows the dispatcher to import a schedule (such as one created by the accompanying MBO module) or create their own, then push the schedule into a queue of trains to be dispatched. The center pane allows the dispatcher to view the train queue and edit schedules as desired, before the train is automatically dispatched or the dispatcher clicks “dispatch now” on a particular train. The right-most pane allows the dispatcher to monitor currently activate trains, including their location, speed, authority, and passengers, as well as modify their speeds, authority, and schedules.

The lower pane that spans the bottom of the frame allows the dispatcher/track maintainer (more on this differentiation in Section 4.1.2) to perform maintenance on tracks, either closing block for maintenance or re-opening previously closed/broken blocks.

4.1.2 USE CASE

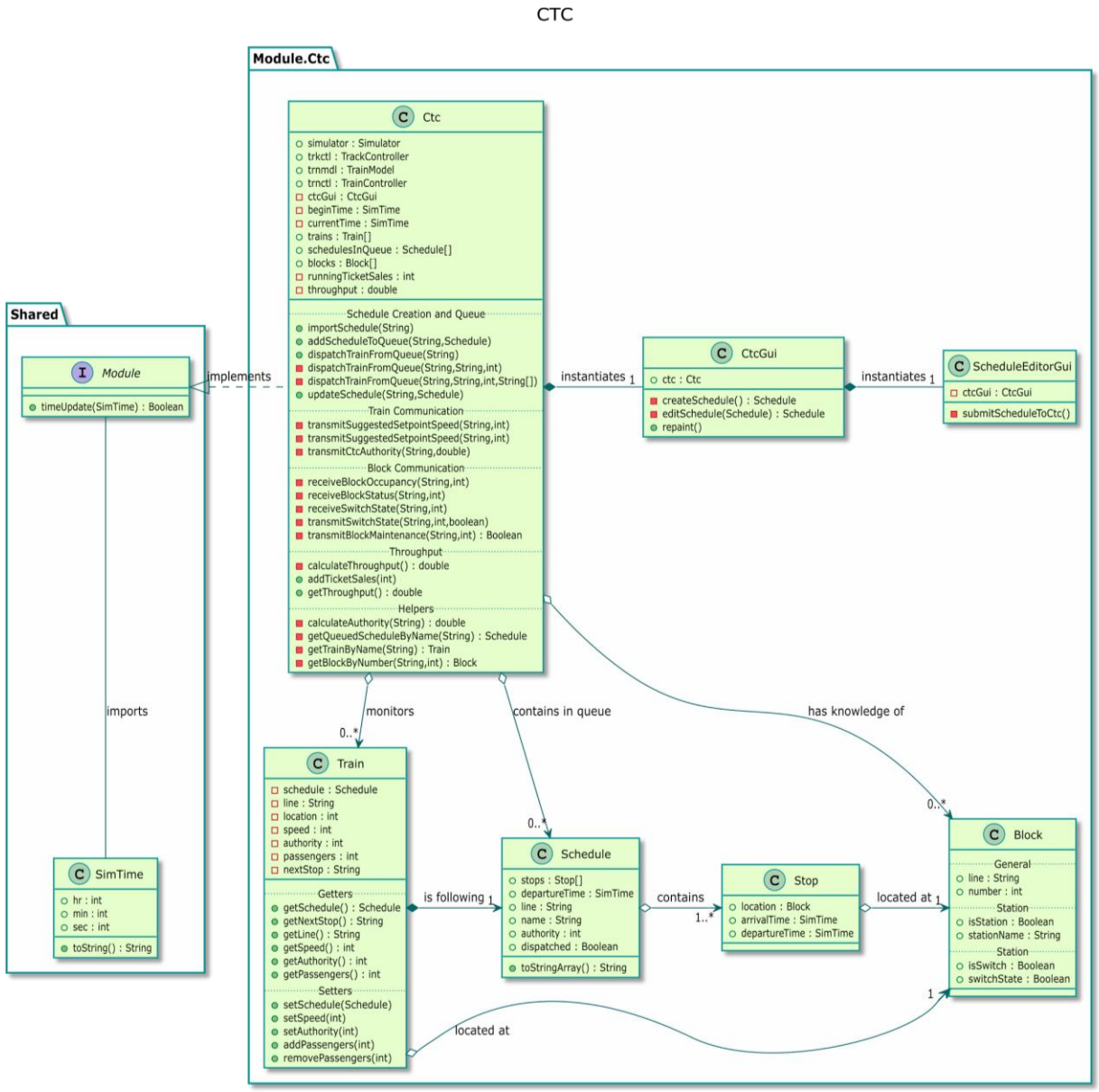


The use cases for the CTC are diagrammed above, and include uses related to schedule creation, train dispatching, train monitoring, and block control. The main user was labelled as “Ctc Officer,” which is essentially one user who has multiple responsibilities including “Dispatcher” and “Maintainer” of the track. In a practical train system, these could be two separate users, but they have been proposed as one user in this particular implementation.

The Officer, acting as the Dispatcher, will have the ability to monitor the train system, import or create schedules, dispatch trains, control trains via suggested speed and setting authority, and toggle switches. The Officer, acting as the Maintainer, can view block failure status and ultimately close or open tracks for trains.

The execution flow of the aforementioned use cases are explicitly rendered in sequence diagrams that can be found in Section 4.1.5.

4.1.3 CLASS DIAGRAM



The class diagram for the CTC is shown above. The main class within the Module.Ctc package is Ctc. This class implements Module, connects to other modules, instantiates the Ctc GUI, and contains the train, schedule, and block object aggregations. Each of these three main functionalities are described below:

- Implements Module - Module is implemented (just as in every other North Shore Extension module) to contractually require the *timeUpdate(SimTime)* method within the Ctc. As described in Section 3, this function is meant to serve as a “rising clock edge” signalling the passage of one virtual second to the module. This function will be used within the Ctc to repaint the gui, recalculate throughput, detect block occupancies across the tracks, and automatically dispatch trains set to be dispatched at the current time. Implementing Module

also imports SimTime which allows the Ctc to automatically dispatch trains on time and calculate time elapsed from the start of the simulation to accurately determine throughput in the calculateThroughput(int) method.

- Connects to other modules - The Ctc class has references to the Simulator, Track Controller, Train Model, and Train Controller. These cross-references will be set by the Simulator immediately after all modules are initialized, as shown in the Simulator state diagram in Section 3. The reasonings for providing direct communication routes to these modules are as such:
    - Simulator - This path will be used to start/pause, as well as control the simulation speed from within the Ctc class. It will also be used to control the simulation weather and temperature from within the Ctc class. These functionalities have yet to be included within the GUI shown in Section 4.1.1.
    - Track Controller - This path will be used to fetch a copy of the Blocks, as well as if they are stations or switches, during initialization of the Ctc. It will be used continually to indirectly communicate with the Track Model to get block occupancy. It will also be used to indirectly transmit authority and suggested speed to the Train Model and Controller.
    - Train Model - This path will be used solely for creating a new train instance when dispatching. Other than this initial instantiation, the Ctc will have no communication with the Train Model.
    - Train Controller - This path will be used to notify the Train Controller that a Train Model has been instantiated with a given name, therefore allowing the Train Controller to actively communicate with the Train Model.
- Instantiates the CtcGui - The Ctc will create a single CtcGui on initialization by the simulator, which will serve as a hub for the Ctc Officer (Dispatcher and/or Maintainer, see Section 4.1.2) to interact with the train system. The CtcGui will have a reference back to the Ctc to communicate with the stored Schedules, Trains, and Blocks (see next bullet on data aggregations). The CtcGui can instantiate the ScheduleEditor GUI, which is an auxiliary GUI meant to aid in Schedule creation and post-creation editing. ScheduleEditor has a reference back to the CtcGui and its public Ctc reference to communicate completed Schedules once the Dispatcher is finished editing.
- Data aggregations - The Ctc class relies on several custom data classes stored in aggregation data structures to monitor the state of the blocks, schedules (un-dispatched trains) and trains.
    - Block - The Block class represents a single block on the track. A list of blocks, including if they are a switch or station, will be fetched from the Track Model into the Ctc at Ctc initialization.
    - Schedule - The Schedule class represents a list of stops and train metadata necessary for a train to function. The list of stops, where each stop corresponds to a block location at a given arrival time, dictates authority throughout the train's journey. Once a Schedule is set to be dispatched, it is merged into a new Train object (as described below).

- Train - The Train class is used for Schedules that have been dispatched to the track. The Train includes a Schedule, as well as a current speed, authority, location, and occupancy.

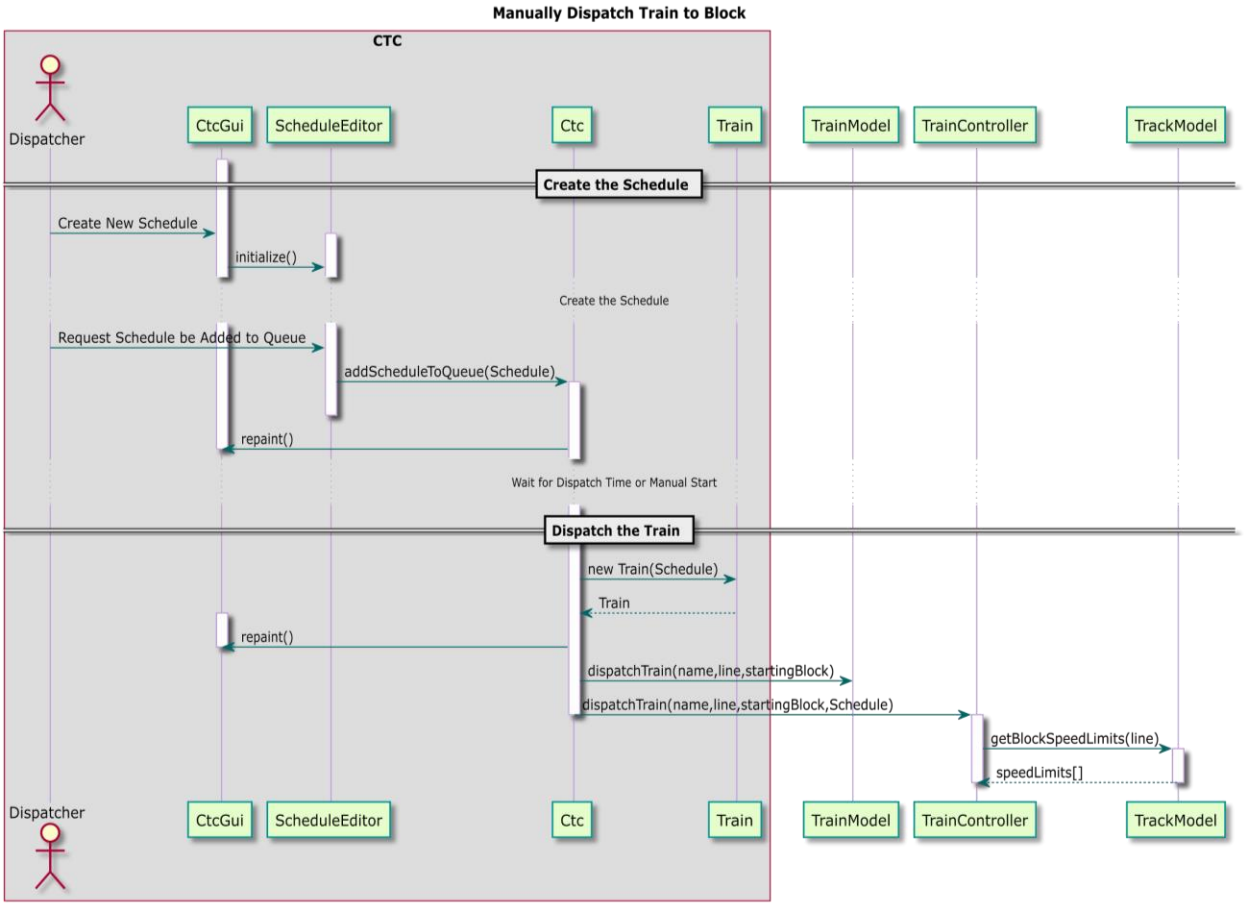
4.1.4 DATA DESIGN

Within the CTC, there are multiple important, aggregate data structures making it necessary to outline the data structures that will be used. A list of aggregated objects and their proposed data structures are as follows:

Data Being Stored	Proposed Data Structure	Rationale
List of Schedule objects waiting to be dispatched	HashMap<String,Schedule>	It is useful to fetch a train Schedule by its name since it will be displayed by name in the GUI table. This way, if a user clicks on a train to analyze the schedule, the program may simply fetch the schedule by the train name shown on the given train's row.
List of active Train objects that have been dispatched	HashMap<String,Train>	It is useful to fetch a Train by its name since it will be displayed by name in the GUI table. This way, if a user clicks on a train to analyze its speed, authority, and schedule, the program may simply fetch the Train by the train name shown on the given train's row.
List of Block objects and their physical properties	Block[ ]	Since blocks are directly addressable with ints, it is useful to store the Block objects in an array of Blocks, ordered such that element 1 has Block 1, and so on.

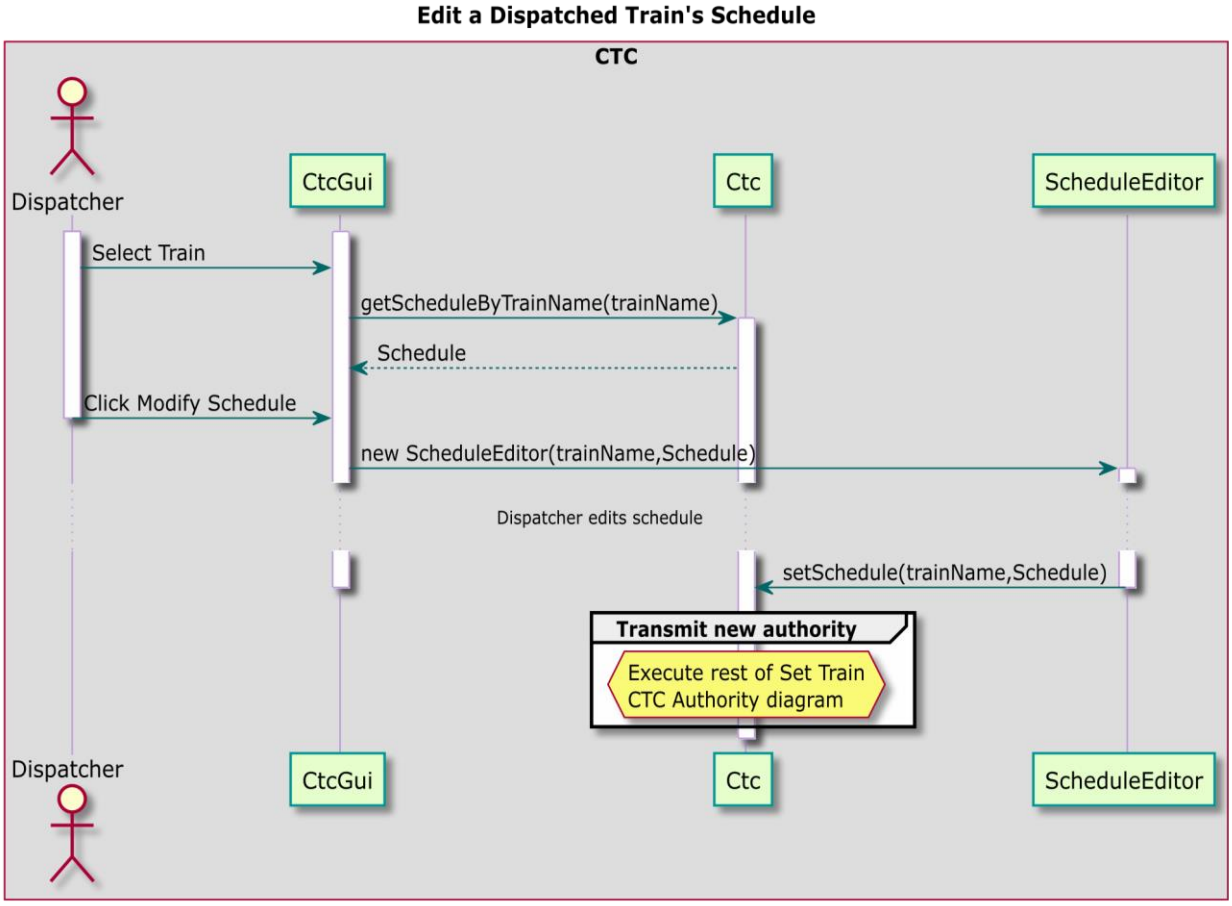
4.1.5 SEQUENCE DIAGRAMS

4.1.5.1 Manually Create and Dispatch Train



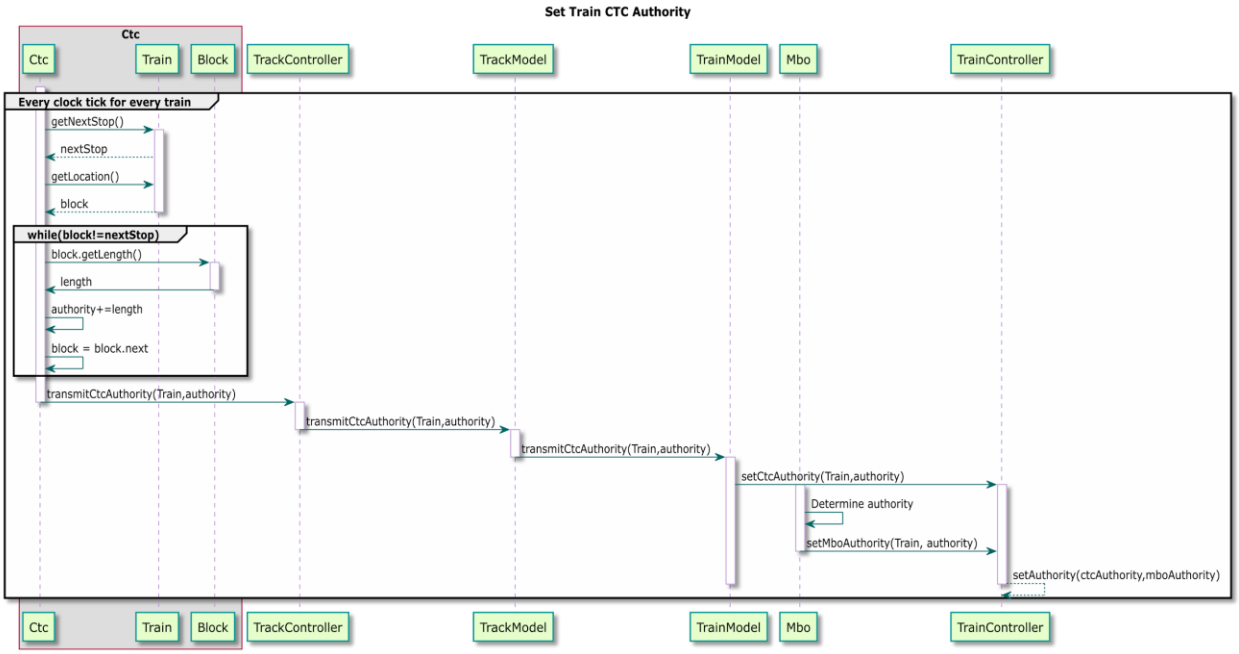
The diagram for manually creating and dispatching a new train (either from the yard or from a block) is shown above. It begins with the Dispatcher expressing their interest to create a new schedule, and clicking the appropriate button on the CtcGui. The GUI opens an auxiliary ScheduleEditor window, where the dispatcher proceeds to create a schedule. Upon completion, the Dispatcher clicks submit, and the schedule is exported from ScheduleEditor to the main Ctc class which promptly shows the schedule in the queue. The sequence waits until the train's dispatch time, or the Dispatcher's manual override to dispatch the train. A Train object is created from the schedule, which is reflected in the GUI. The Ctc then issues a dispatchTrain() command to both the Train Model (to create a new TrainModel object) and the Train Controller (to notify it that a new train exists with the given name).

4.1.5.2 Edit Dispatched Train's Schedule



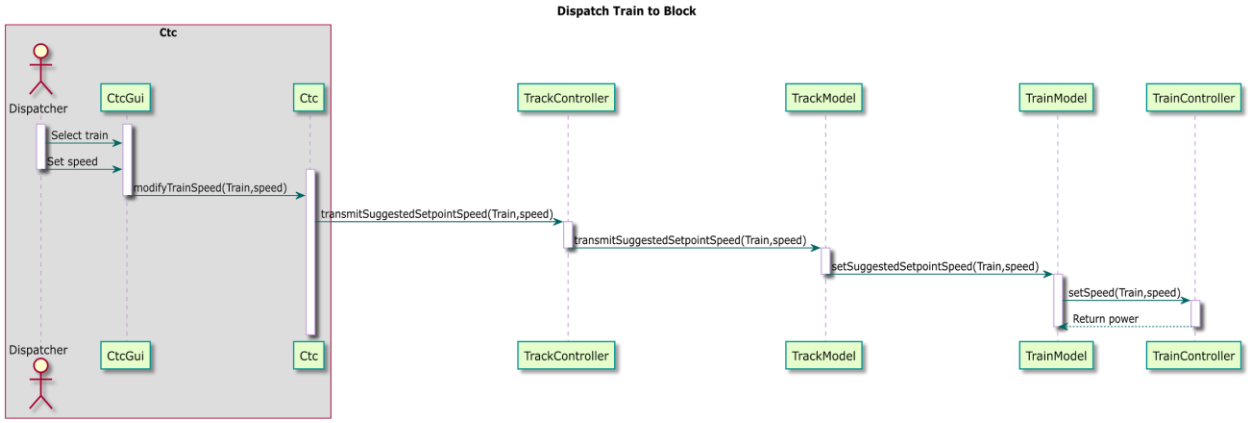
The diagram for editing a dispatched train's schedule is shown above. It begins with the Dispatcher selected the train from a table of dispatched trains within the GUI. This fetches the train's current schedule for display within the GUI. The Dispatcher then clicks an "Edit Schedule" button, prompting the auxiliary ScheduleEditor window to pop up, already prefilled with the current schedule. The Dispatcher edits the schedule and clicks confirm which sends the updated schedule back into the main Ctc class. Now, the Ctc must run the subroutine for transmitting a new authority, which is diagrammed in Section 4.1.5.3.

4.1.5.3 Set Train Authority



The diagram for transmitting an authority is shown above. The calculation of train authority occurs for every train at every virtual second of the simulation and therefore resides within the updateTime(SimTime) function shown in the CTC class diagram (Section 4.1.3). It begins with the Ctc fetching a given train's next stop, as well as current location from an internal copy of the Train object. It then traverses subsequent blocks, cumulatively summing the block lengths to find the distance to the next stop. This distance to the next stop is the authority that is then transmitted to the Track Controller, who aids in propagating the authority to the Train Controller.

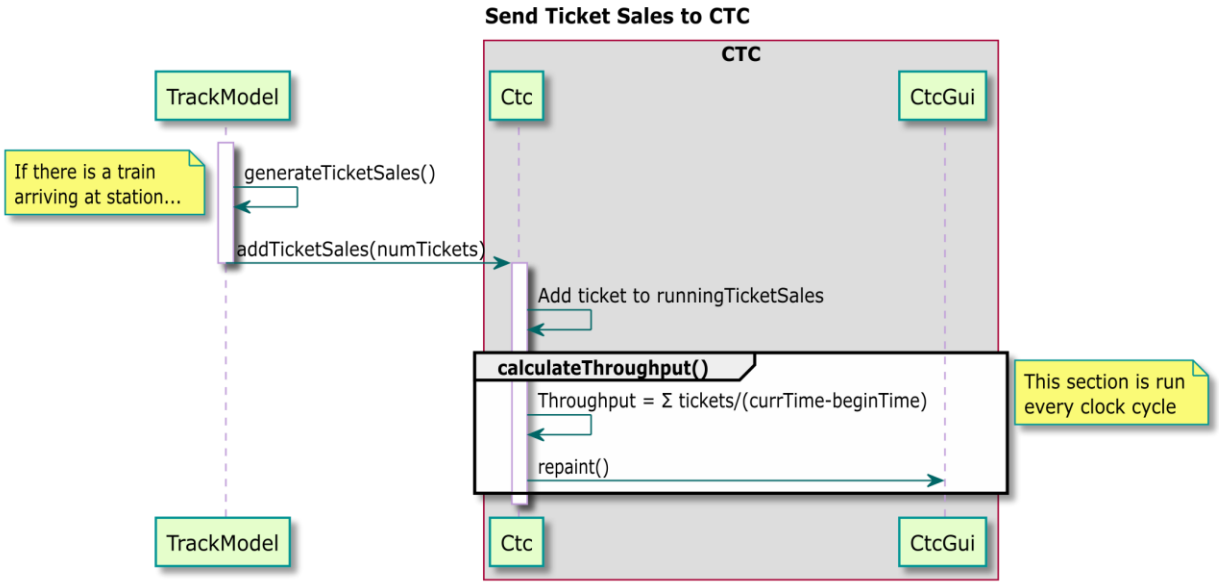
4.1.5.4 Suggest Setpoint Speed



The diagram for transmitting a suggested speed to a train is shown above. It begins with the Dispatcher selecting a train from from a table of dispatched trains, then modifying its speed within the GUI. This triggers a transmission to the Ctc, telling it to transmit the new speed to the Train Controller as a suggestion. The Ctc class transmits

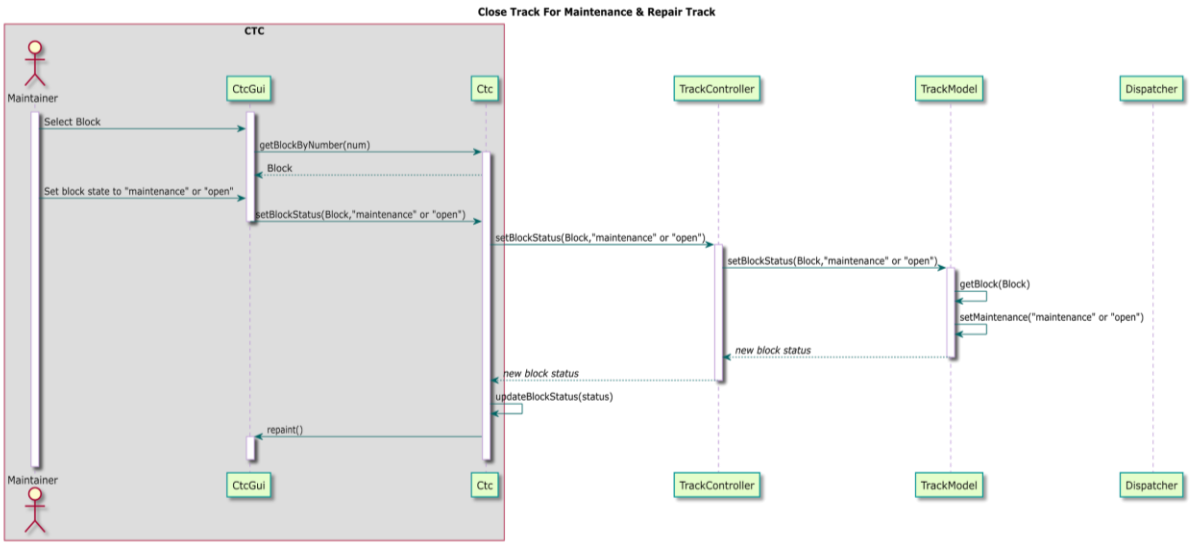
the speed to the Track Controller, who then aids in propagating the suggested speed to the Train Controller.

4.1.5.5 Add Ticket Sales from Track Model



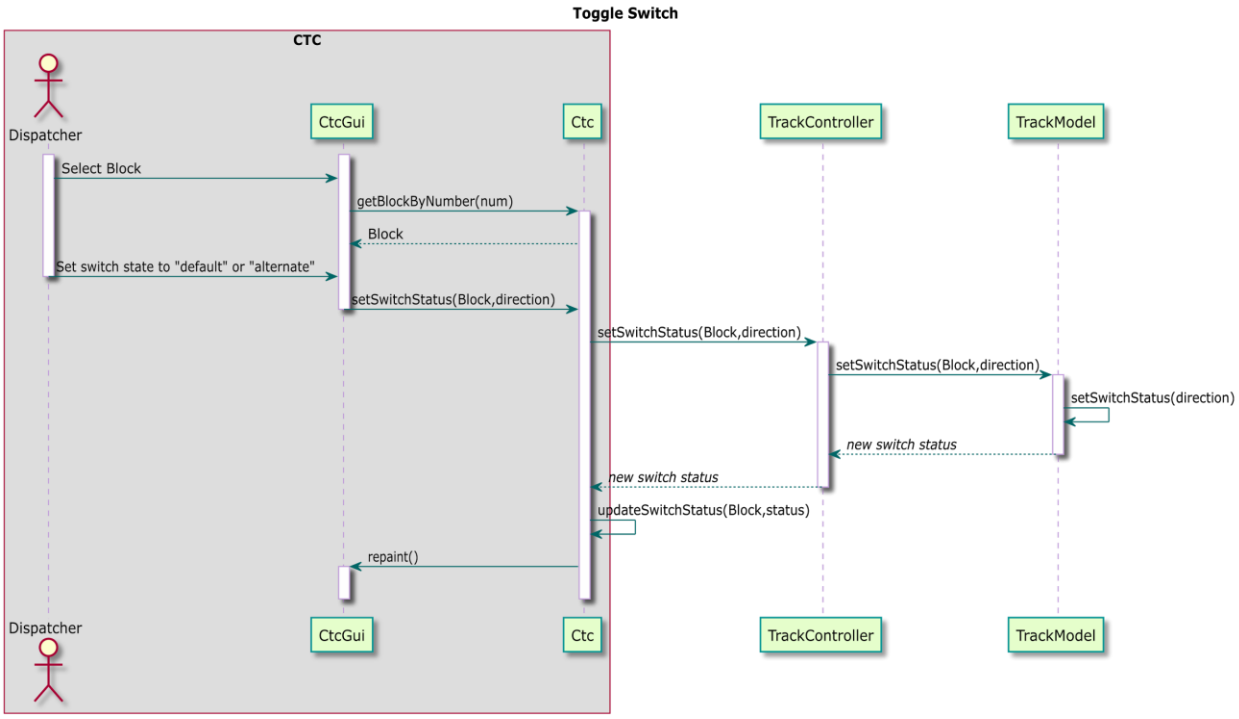
The diagram for receiving ticket sales and recalculating throughput is shown above. It is important to note that throughput is automatically recalculated at every virtual second of the simulator, and therefore is called within the updateTime(SimTime) function shown in the CTC class diagram (Section 4.1.3). However, the functionality may also be called ad hoc, such as immediately after ticket sales are received (as shown in the diagram above). The process begins with the Track Model sending the Ctc a number of tickets sold for a recent train. The tickets are added to a grand total within the Ctc, and throughput is calculated as the number of tickets sold over the elapsed virtual time of the simulation. Once calculated, the CtcGui is updated to reflect the new throughput value.

4.1.5.6 Perform Track Maintenance



The diagram for updating the status of a block to either “open” or “maintenance” is shown above. It was deemed appropriate to condense the functionalities of closing a block for maintenance and reopening a closed block since they share the same sequence flow. First, the Maintainer (see Section 4.1.2 for users reference) selects a block using the GUI. Upon doing so, the GUI fetches the block from the Ctc for status display. The Maintainer can then choose what to update the track’s status to - they can reopen a closed or failed block, or mark a block as closed for maintenance. Upon doing so, the new block status is propagated through the Track Controller to the Track Model, who then replies a confirmation of the new block status. The Ctc receives this confirmation, updates its internal copy of the blocks to reflect it, and updates the GUI accordingly.

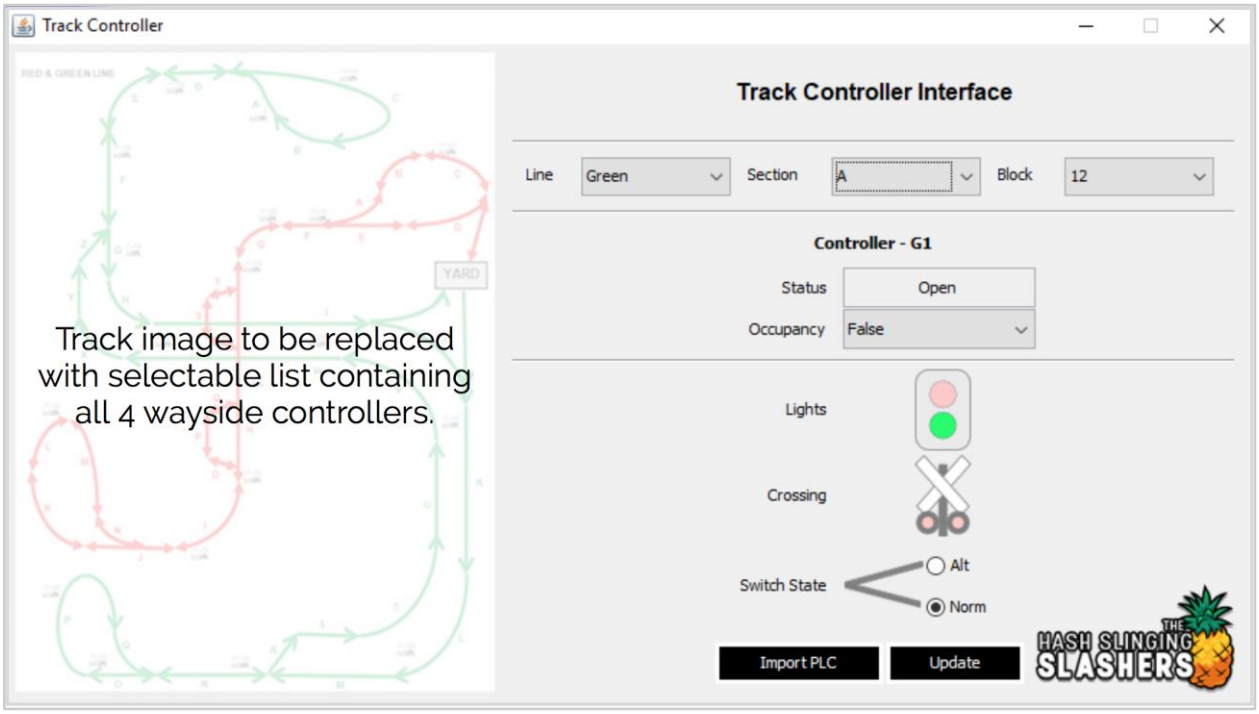
4.1.5.7 Toggle Switch



The diagram for toggling a switch (or explicitly setting it to its default or alternate state) is shown above. First, the Dispatcher selects a block using the GUI, prompting the GUI to fetch the block info and display it. The Dispatcher then expresses what they would like the switch state to be. Upon doing so, the new switch state is propagated through the Track Controller to the Track Model, who then replies a confirmation of the new switch state. The Ctc receives this confirmation, updates its internal copy of the switch to reflect it, and updates the GUI accordingly.

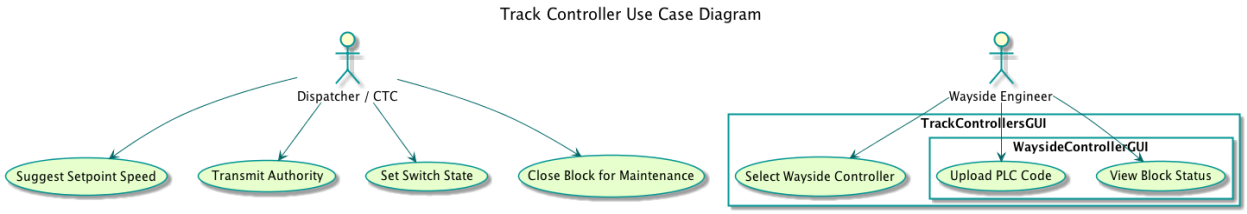
4.2 TRACK CONTROLLER

4.2.1 INTERFACE



The above GUI serves as a reminder of the Track Controller user interface. The GUI is navigated from left to right, and top to bottom. The left panel is going to be replaced with a list of all four wayside controllers, allowing the user to select which wayside controller they wish to look at or interact with. Selecting a wayside controller will populate the right panel with the information about that specific wayside. From there the user can select a block controlled by that wayside to view the status of the block and states of the hardware on the block. At the bottom, the user can upload PLC code by clicking the button labeled *"Import PLC"* and then manually force the controller to update the state of the hardware by clicking the *"Update"* button, however the update functionality may be phased out in a future iteration of the GUI. Lastly the user can manually change the state of hardware by clicking on the hardware icon or buttons, however this is only on blocks that contain the hardware shown.

4.2.2 USE CASE



The use cases for the Track Controller are diagrammed above, and include uses related to transmitting setpoint speed and authority, as well as uploading PLC and controlling

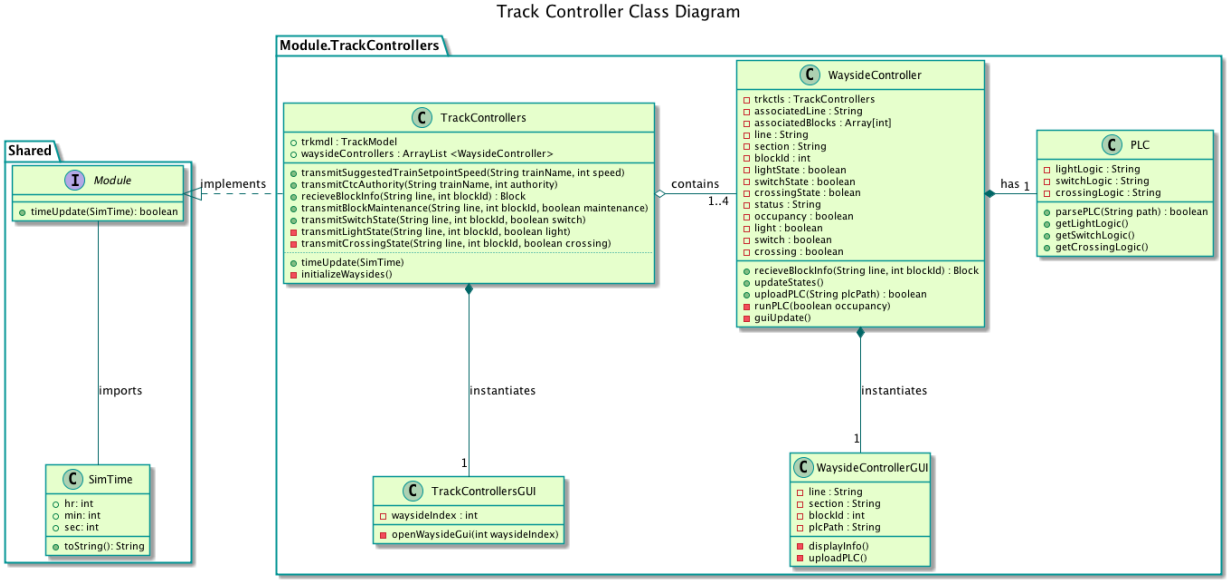
block hardware. The main user is labelled as “Wayside Engineer,” and the other is a “Dispatcher” which sends commands through the CTC that need attended to.

The Wayside Engineer will have the ability to select an individual Wayside Controller from the Track Controllers GUI which will then present them with the GUI for the associated Wayside Controller. From there they can upload PLC code to that Wayside Controller or view the block status of an associated block.

The Dispatcher, residing at the CTC can suggest a setpoint speed or transmit an Authority both of which are then passed on to the Track Model to be passed along further. They can also toggle a switch state manually of a switch on the track or close a block for maintenance or repair, both of which are handled by the individual Wayside Controllers.

The execution flow of the aforementioned use cases are explicitly rendered in sequence diagrams that can be found in Section 4.2.5.

4.2.3 CLASS DIAGRAM



Excluding the shared section of the class diagram which is implemented by every module, the main class for this module is called Track Controllers. The Track Controllers class has all of the functions used to pass information to other modules as well as some private ones for self-use. Mainly, the Track Controllers class transmits is the setpoint speed and authority from the CTC, as well as toggling a switch or track maintenance, which can be initiated by the CTC. There will only be one instance of the Track Controllers class and it will have a GUI class instantiated which will be the window into which the frames for each Wayside Controller GUI will be placed. The Track Controllers Class other main function is to initialize four Wayside Controller objects, and keep them in an ArrayList. Next the Wayside Controller Class holds all of the main functionality of a Wayside Controller, having fields for all of the associated and pertinent block information and the methods to effectively utilize the PLC class to do the job of the Wayside Controller. Each Wayside Controller class has a PLC object that is able to

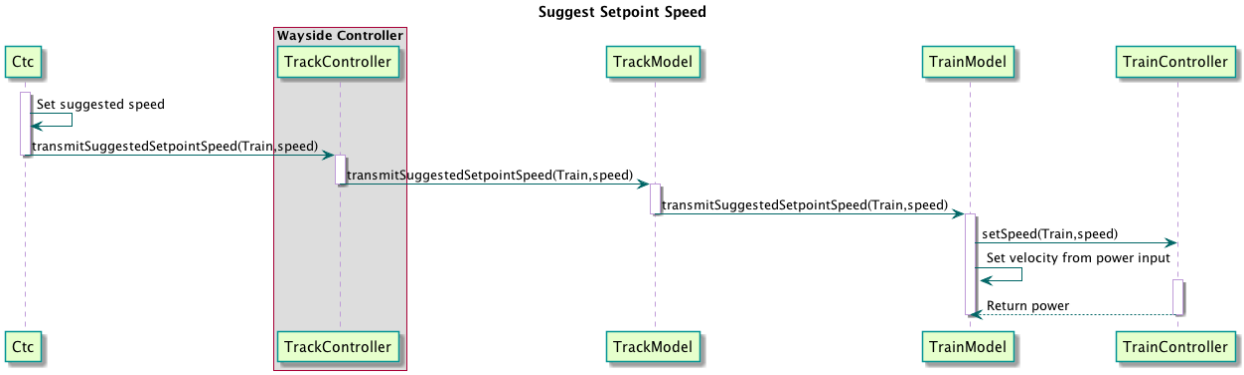
parse and store the logic from the PLC code for use by the Wayside Controller when determining the states of the track hardware. Lastly, the Wayside Controller GUI class has the functions that are necessary for the Wayside Engineer to interact with to perform his/her duties including uploading PLC code and viewing the status of the track hardware. Each Wayside Controller has one GUI object which is the frame placed inside the Track Controllers GUI.

4.2.4 DATA DESIGN

Data Being Stored	Proposed Data Structure	Rationale
Collection of Wayside Controller Objects	ArrayList<WaysideController>	It will be easy to search through the ArrayList for the Wayside Controller object with a particular identifier integer, and then perform operations with that specific Wayside.
Collection of associated Blocks	Array[int]	This field will contain the list of associated Block indices for each Wayside Controller for the drop-down menus presented to the Engineer.

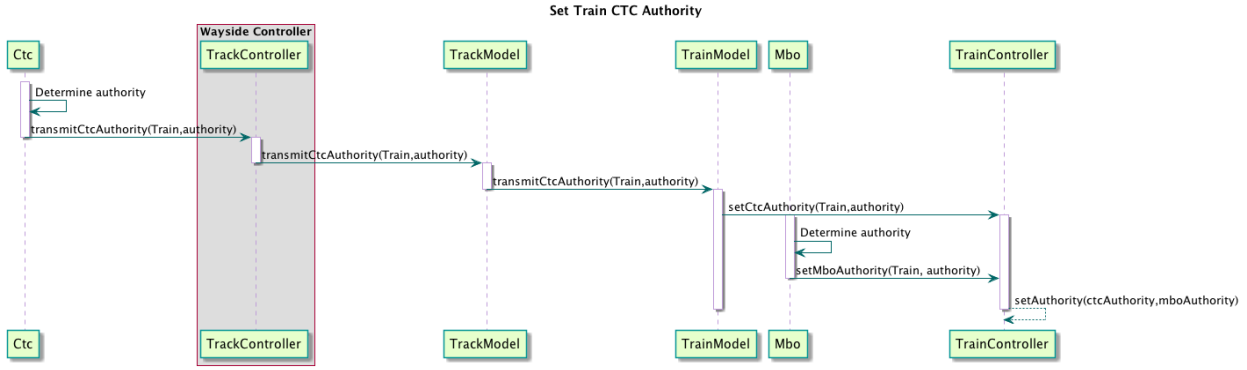
4.2.5 SEQUENCE DIAGRAMS

4.2.5.1 Suggest Setpoint Speed - Track Controller



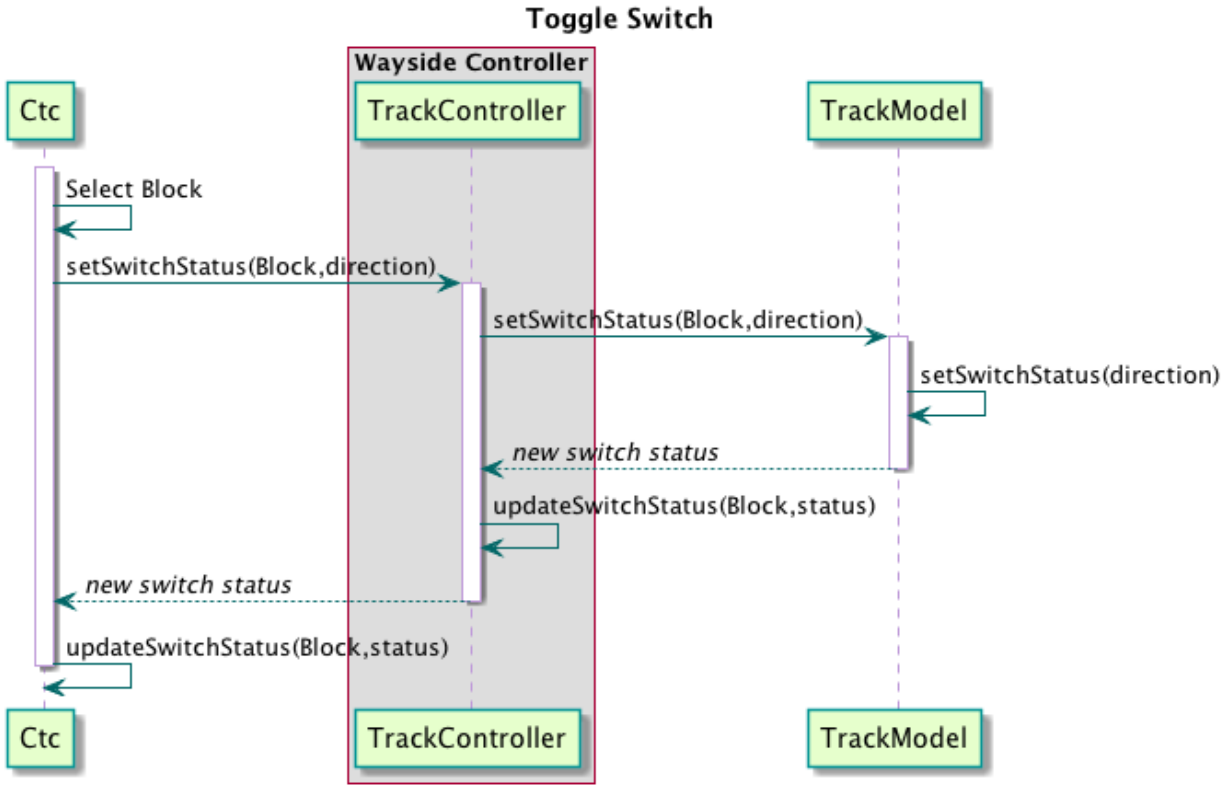
Shown in this diagram is the transmission of the suggested setpoint speed by the CTC to the Train Controller. The involvement of the Wayside Controller is shown in the highlighted box, but in summary just accepts the incoming signal and passes it along to the Track Model by calling a function within the Track Model class.

4.2.5.2 Transmit Authority - Track Controller



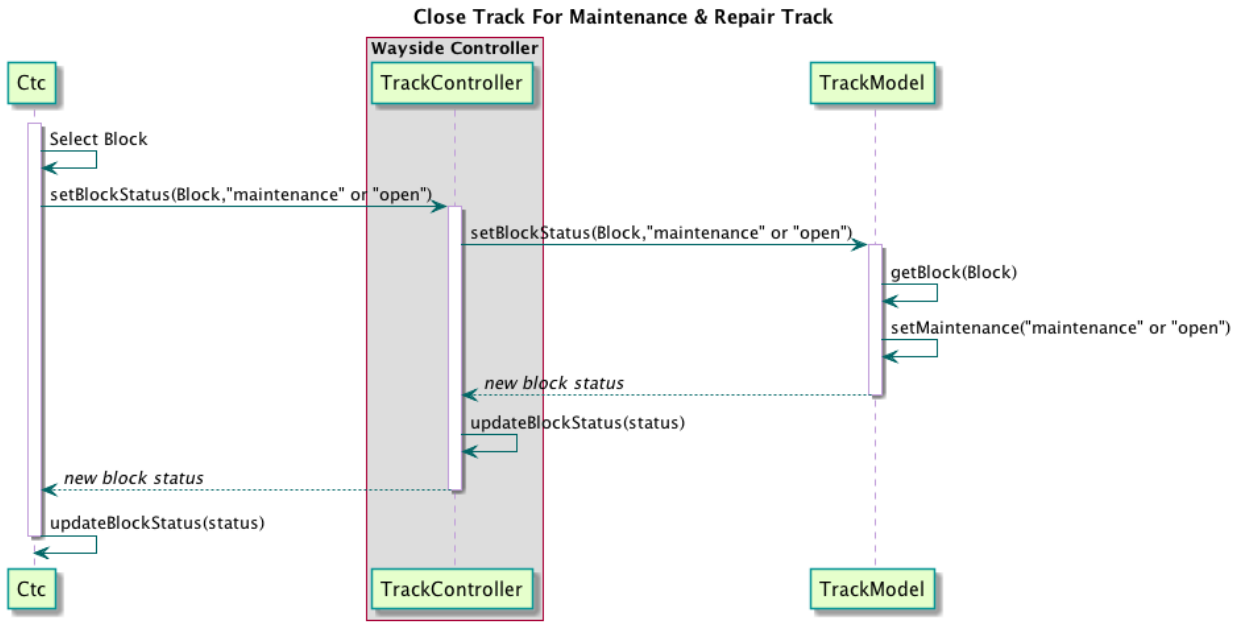
Shown in this diagram is the transmission of the authority set by the CTC to the Train Controller. The involvement of the Wayside Controller is shown in the highlighted box, but in summary just accepts the incoming signal and passes it along to the Track Model by calling a function within the Track Model class.

4.2.5.3 Set Switch State



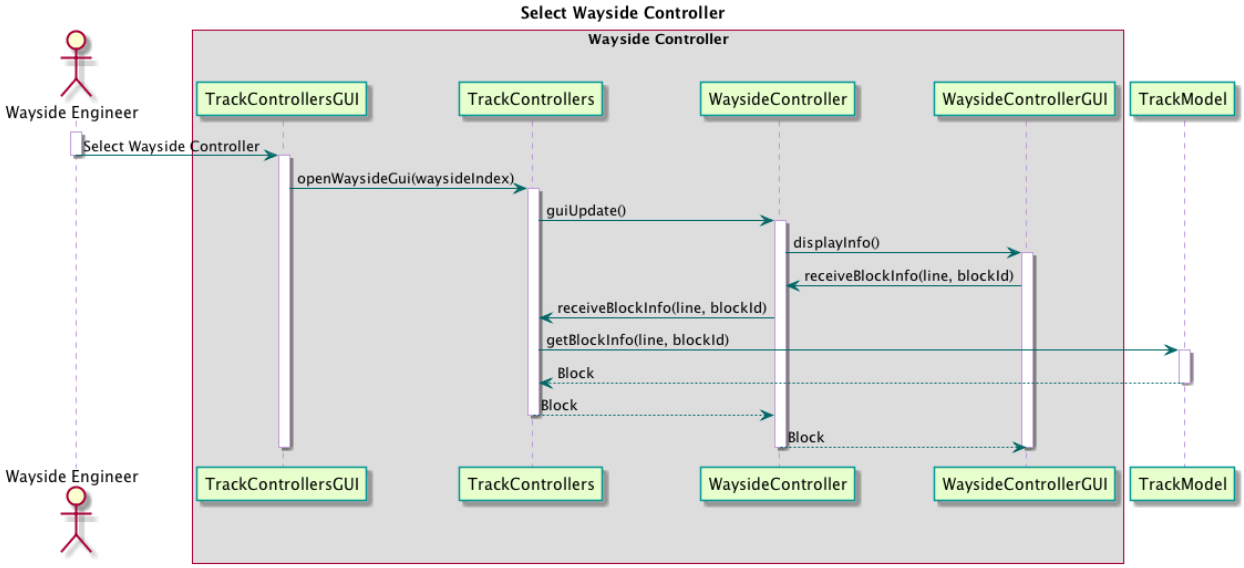
Shown in this diagram is the toggling of a switch state by the CTC. The involvement of the Wayside Controller is shown in the highlighted box, but in summary accepts the incoming signal for toggling the state and passes it along to the Track Model by calling the same function within the Track Model class that is used when the Wayside Controller determines the state of any switch. The signal is first processed to ensure it is a valid toggling state before passing along the signal to the Track Model. The state is then updated within the frame of reference of the Wayside Controller if it is selected.

4.2.5.4 Close Block for Maintenance



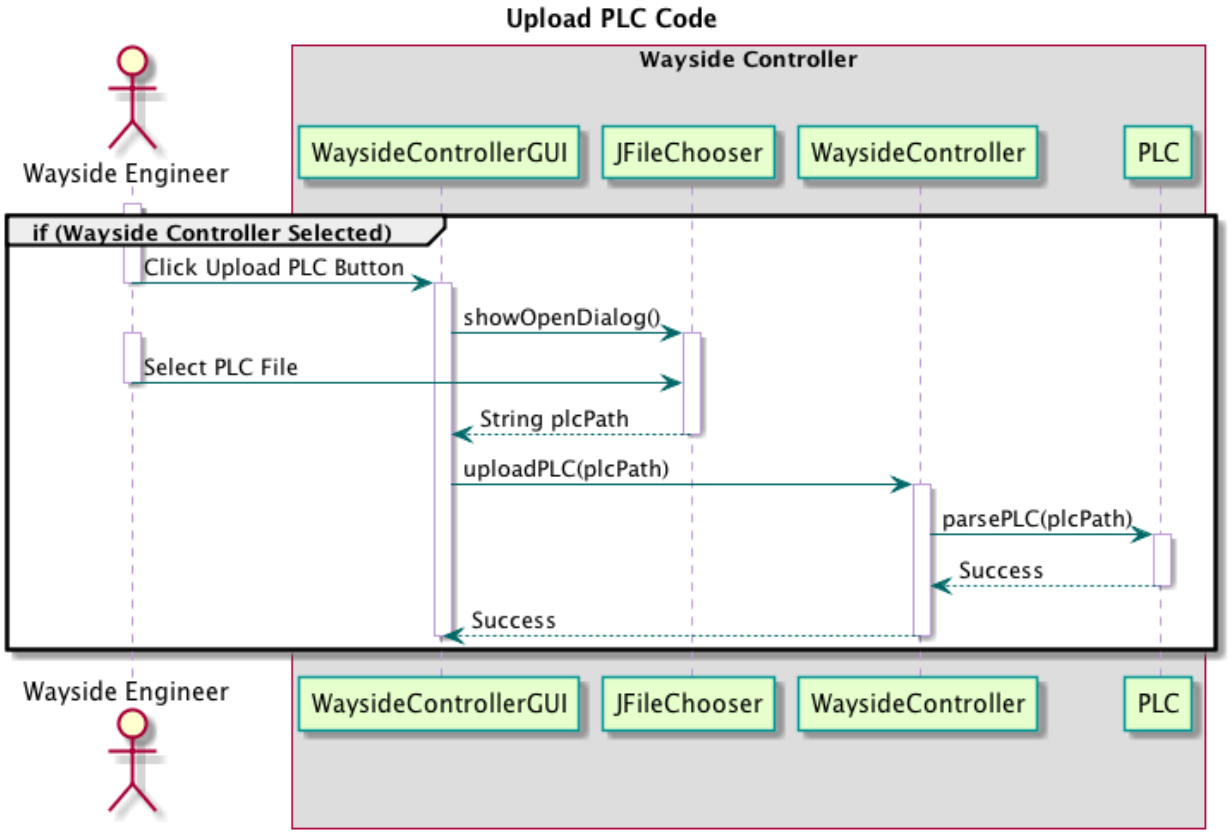
Shown in this diagram is the closing of a block for maintenance or repair by the CTC. The involvement of the Wayside Controller is shown in the highlighted box, but in summary just accepts the incoming signal and passes it along to the Track Model by calling a function within the Track Model class. This request is not vetted by the Wayside Controller as the CTC has the ability to close or change a block status at any time for any reason.

4.2.5.5 Select Wayside Controller



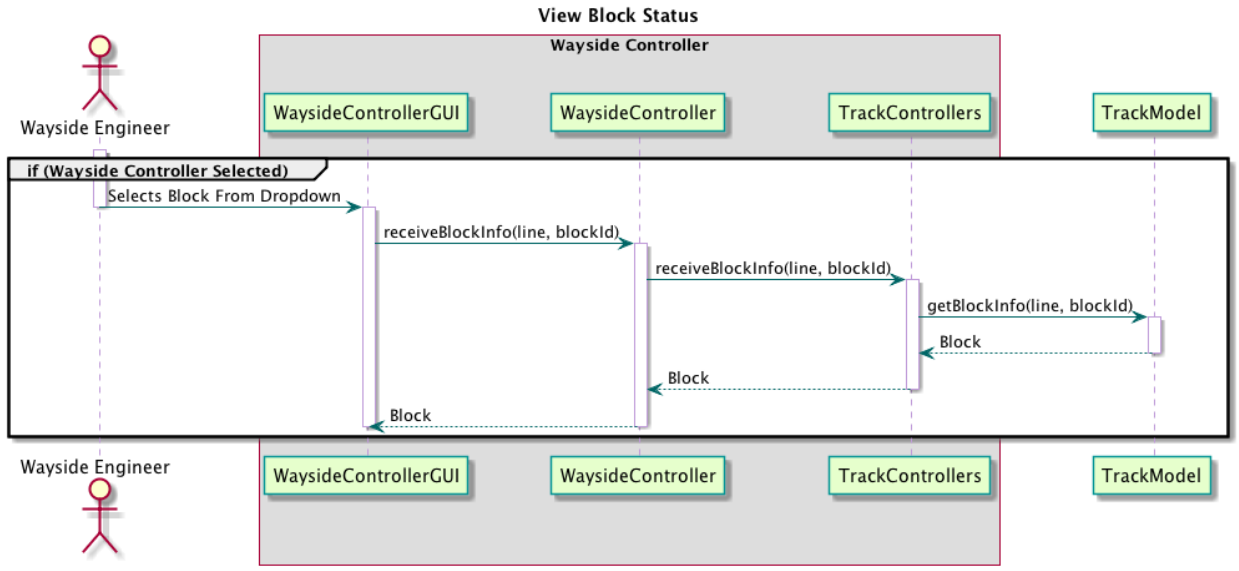
Shown in this diagram is the selection of a Wayside Controller by the Wayside Engineer. The involvement of the Wayside Controller is shown in the highlighted box, but in summary upon the action of the engineer selecting a Wayside from the Track Controllers GUI, a signal is sent through to get the associated Controller's GUI and present the GUI along with the first associated block for the chosen Wayside Controller.

4.2.5.6 Upload PLC Code



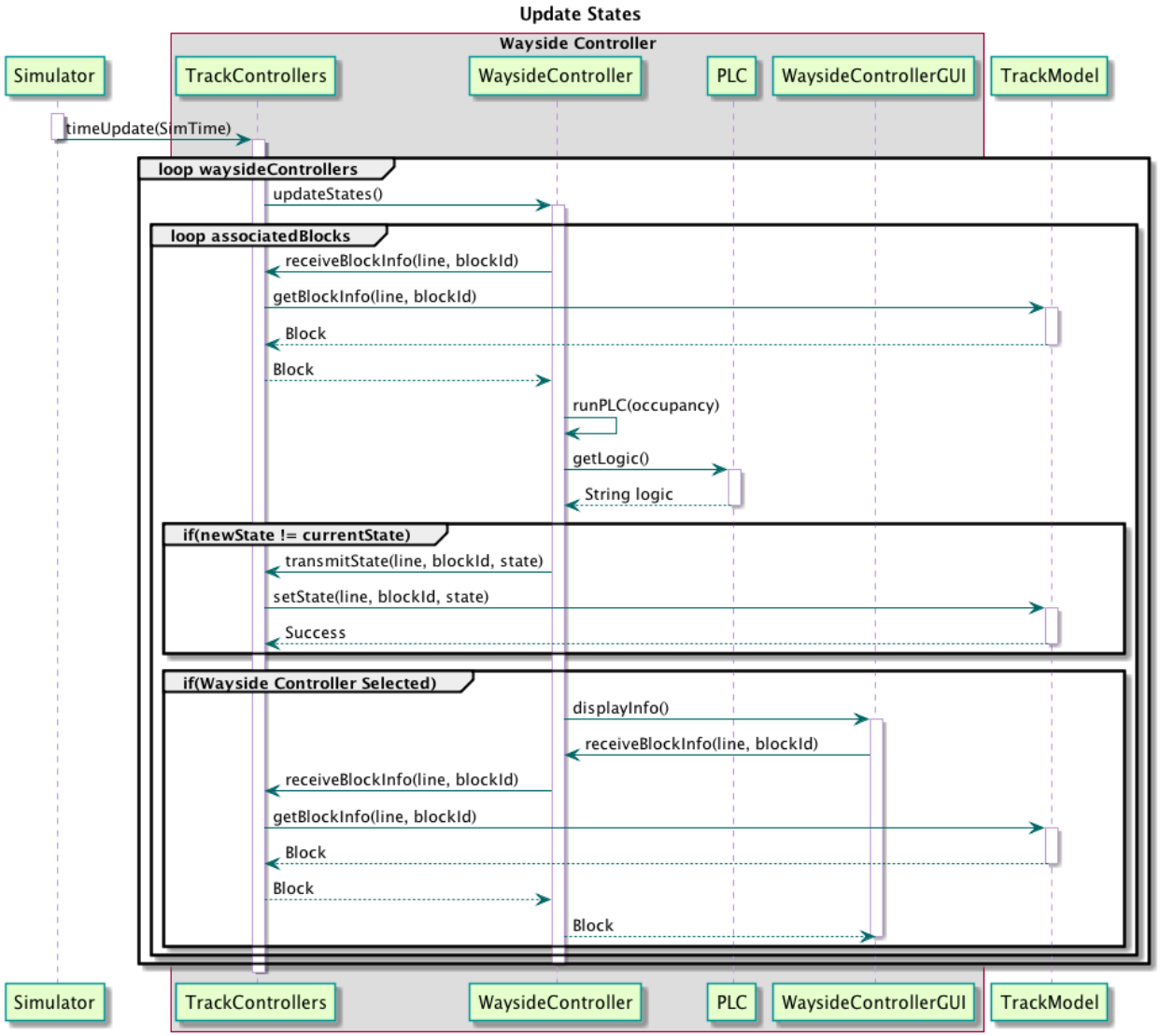
Shown in this diagram is the process of uploading PLC code to a selected Wayside Controller. The involvement of the Wayside Controller is shown in the highlighted box, but it first assumes that a Wayside Controller is selected as shown by the grouping. Upon the clicking of the “Upload PLC” button on the Wayside Controller GUI, the GUI then opens a file choosing window which the engineer can use to choose their file. From there, the path to the file is sent to the PLC object within the Wayside Controller and is parsed for the logic within the proprietary PLC code syntax. A ‘Success’ signal in the form of a boolean is returned and displayed to the engineer.

4.2.5.7 View Block Status



Shown in this diagram is the viewing of a block's status. The involvement of the Wayside Controller is shown in the highlighted box, but it first assumes that a Wayside Controller is selected as shown by the grouping. From there the Wayside Engineer uses the drop-down menus to select an associated block for which they wish to view the status. Once the block is selected, the Wayside Controller sends out a retrieve signal to the Track Model to get the necessary information about the block. After the information is received, the appropriate information is displayed to the engineer. A representation of this is shown in section 4.2.1.

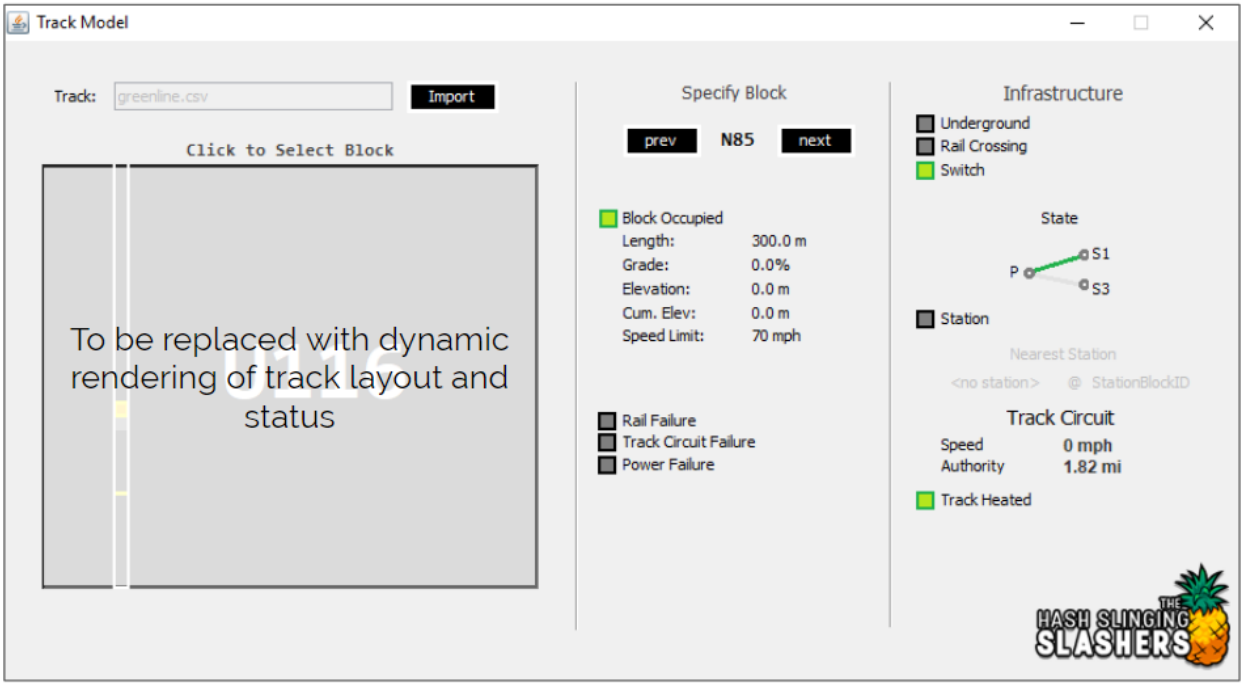
4.2.5.8 Update States



Shown in this diagram is the updating of track hardware states by the Wayside Controller. The involvement of the Wayside Controller is shown in the highlighted box, but the initiating of the sequence begins with the timeUpdate() called by the Simulator, effectively acting as time passing or a clock cycle in engineering terms. From there the Track Controller loops through the Wayside Controllers as shown by the outermost grouping. From there each Wayside Controller loops through its associated blocks and collects the current states of the hardware as well as block occupancy. It then determines if the new state, after running the collected information through the PLC logic, is different than the current state and if so updates the state as stored within the Track Model and otherwise does not do anything. In addition, if a Wayside Controller is selected, then the block information present on the GUI will be updated with the newest information all pulled from the Track Model.

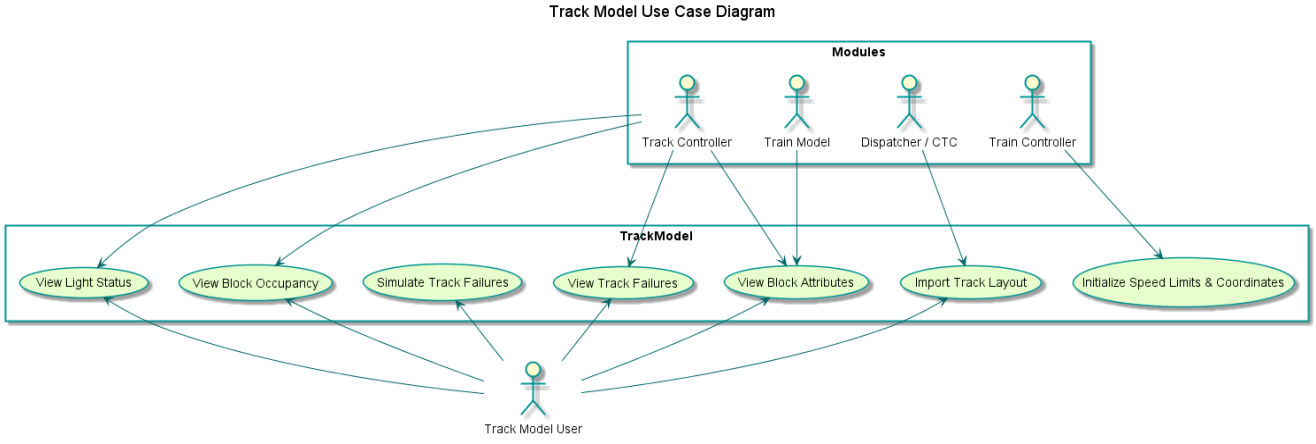
4.3 TRACK MODEL

4.3.1 INTERFACE



The above GUI serves as a reminder of the Track Model user interface. The left pane allows the Track Model user to import a track layout specified in a .csv file. Below the track import area is the dynamic track viewing and selection window which allows a user to view the real-time status of a track and select specific blocks to view their characteristics. The center pane allows the track user to navigate blocks for characteristic viewing as well. Both the center and right panes contain information about block characteristics including static geographical characteristics and dynamic characteristics such as switch and crossing state. Additionally, rail, track circuit, and power failures may be simulated and corrected from this interface.

4.3.2 USE CASE



The use cases for the Track Model are diagrammed above, and includes uses related to track importing and initialization, track status monitoring, and track failure identification.

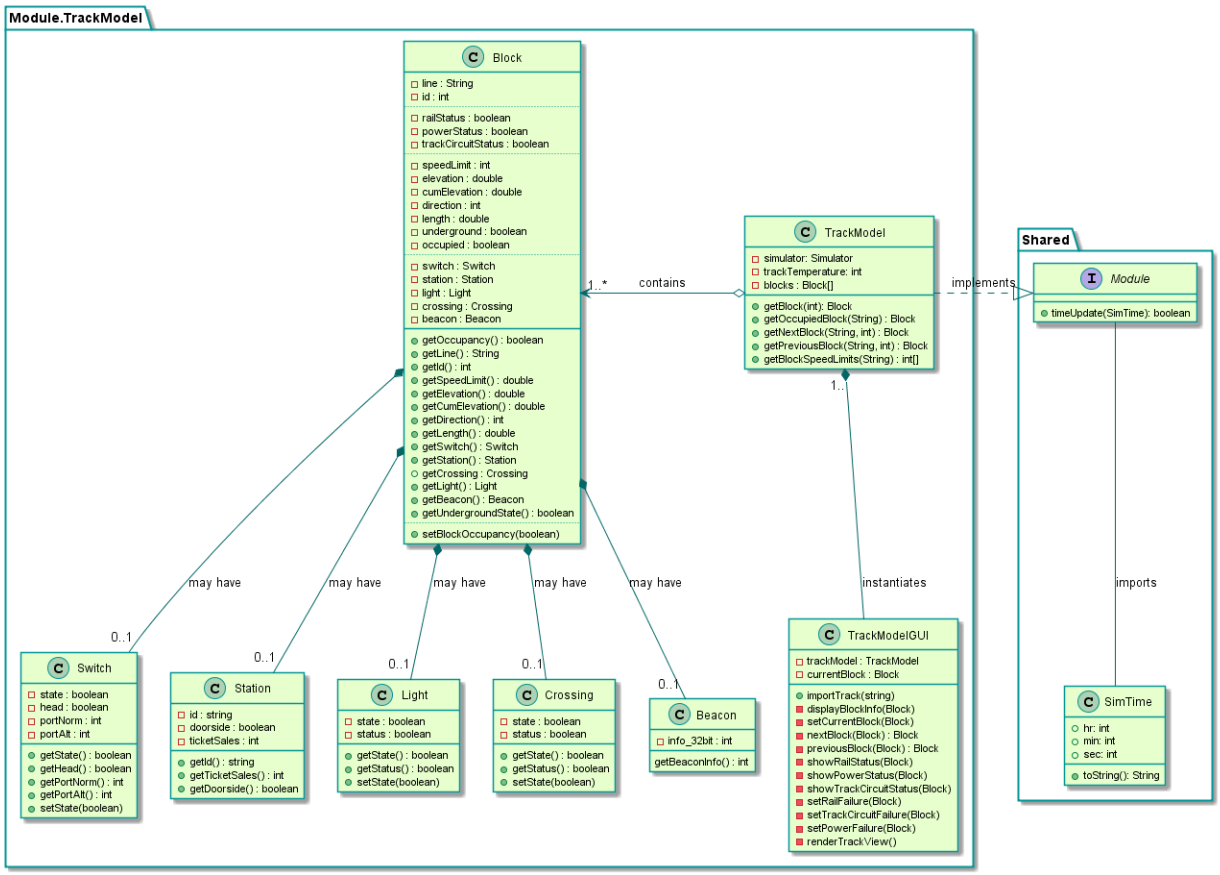
These cases are based primarily on modules that rely on the Track Model's functionality for configurability and display of a track layout.

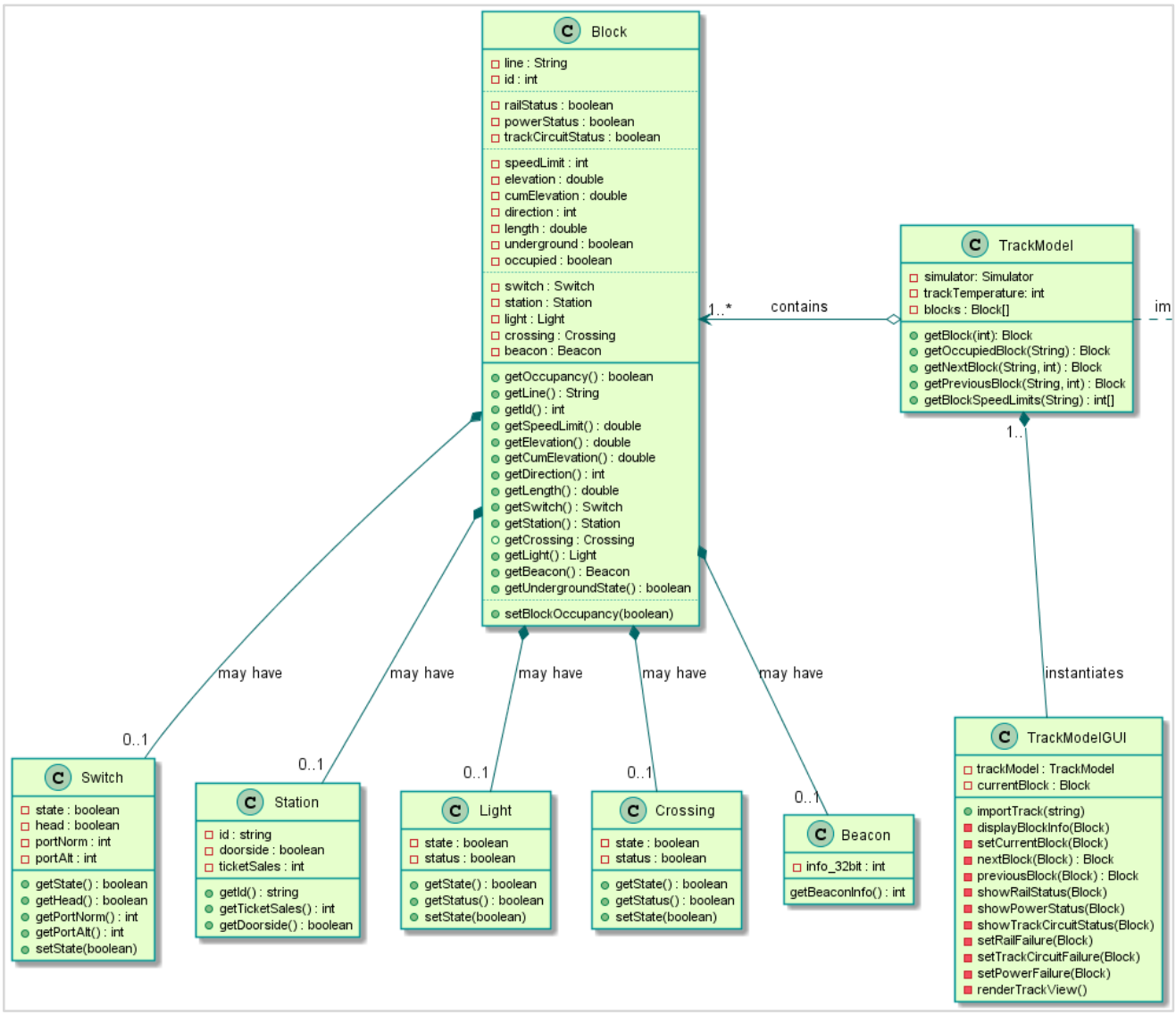
A GUI user that may also use the module for specifying track layout and viewing any track features may be considered a general *Track Model user*. The Track Model user may additionally simulate track failures in the form of power, track circuit, and rail failures for specific blocks.

The execution flow of the aforementioned use cases are explicitly rendered in sequence diagrams that can be found in Section 4.3.5.

4.3.3 CLASS DIAGRAM

Track Model Class Diagram





The class diagram for the Track Model is shown above (zoomed in, not showing *Module* and *SimTime* classes). The main class within the Module.TrackModel package is TrackModel. This class implements Module, connects to other modules, instantiates the Track Model GUI, and contains the aggregations of block objects. Each of these three main functionalities are described below:

- Implements Module - Module is implemented (just as in every other North Shore Extension module) to contractually require the *timeUpdate(SimTime)* method within the Track Model. As described in Section 3, this function is meant to serve as a “rising clock edge” signalling the passage of one virtual second to the module. This function will be used within the Track Model to repaint the track viewing interface and update the status, attributes, and occupancy of blocks.
- Connects to other modules - The Track Model class has references to the CTC, Track Controller, and Train Model main classes. These cross-references will be set by the Simulator immediately after all modules are initialized, as shown in the

Simulator state diagram in Section 3. The reasonings for providing direct communication routes to these modules are as such:

- Simulator - This path will be used to start/pause, as well as control the simulation speed from within the Track Model class. It will also be used to access the simulation weather and temperature from within the Track Model class. These functionalities have yet to be included within the GUI shown in Section 4.1.1.
- CTC - This path will be used to transmit ticket sales received from trains passing through stations.
- Track Controller - This path will be used to transmit a copy of the Blocks, as well as if they are stations or switches, during initialization of the Track Controller module. It will be used continually to indirectly communicate with the Track Controller to transmit track status and recieve updates to the track status from the Track Controller. It will also be used to indirectly recieve authority and suggested speed from the Train Controller to the Track Model.
- Train Model - This path will be used to communicate track characteristics and track status to the Train Model in order for the Train Model to consider physical factors for train movement as well as allowable directions of travel.
- Train Controller - This path will be used for the Train Controller to initialize track layout information at the start of the simulation, particularly block speed limit and coordinates corresponding to locations along the track with a specified resolution within 1 meter.

- Instantiates the Track Model GUI - The Track Model will create a single TrackModelGUI on initialization by the simulator, which will serve as a hub for the Track Model user to interact with the train system. The TrackModelGui will have a reference back to the Track Model to communicate with the track status and block characteristics.
- Data aggregations - The Track Model class relies on a custom data class stored in an aggregation data structure to monitor the state of the blocks that comprise a track layout.
  - Block - The Block class represents a single block on the track and characteristics inherent to that block, including geographical / physical characteristics, occupancy, and the status / availability of switches, lights, crossings, and stations.

4.3.4 DATA DESIGN

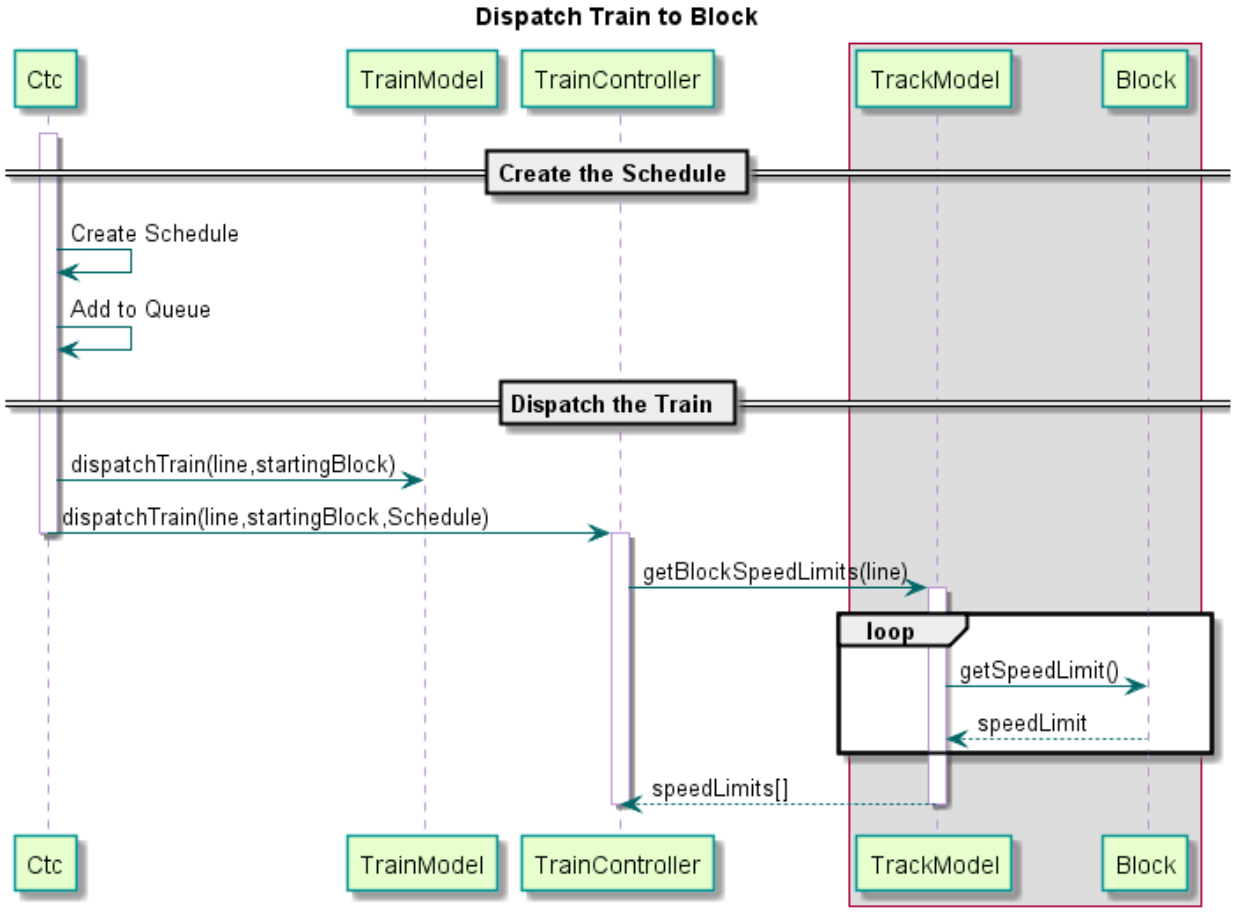
Data Being Stored	Proposed Data Structure	Rationale
Collection of Block Objects	ArrayList<Block>	It will be easy to search through the ArrayList for the Block object with a particular identifier integer,

		and then perform operations on that specific block.

4.3.5 SEQUENCE DIAGRAMS

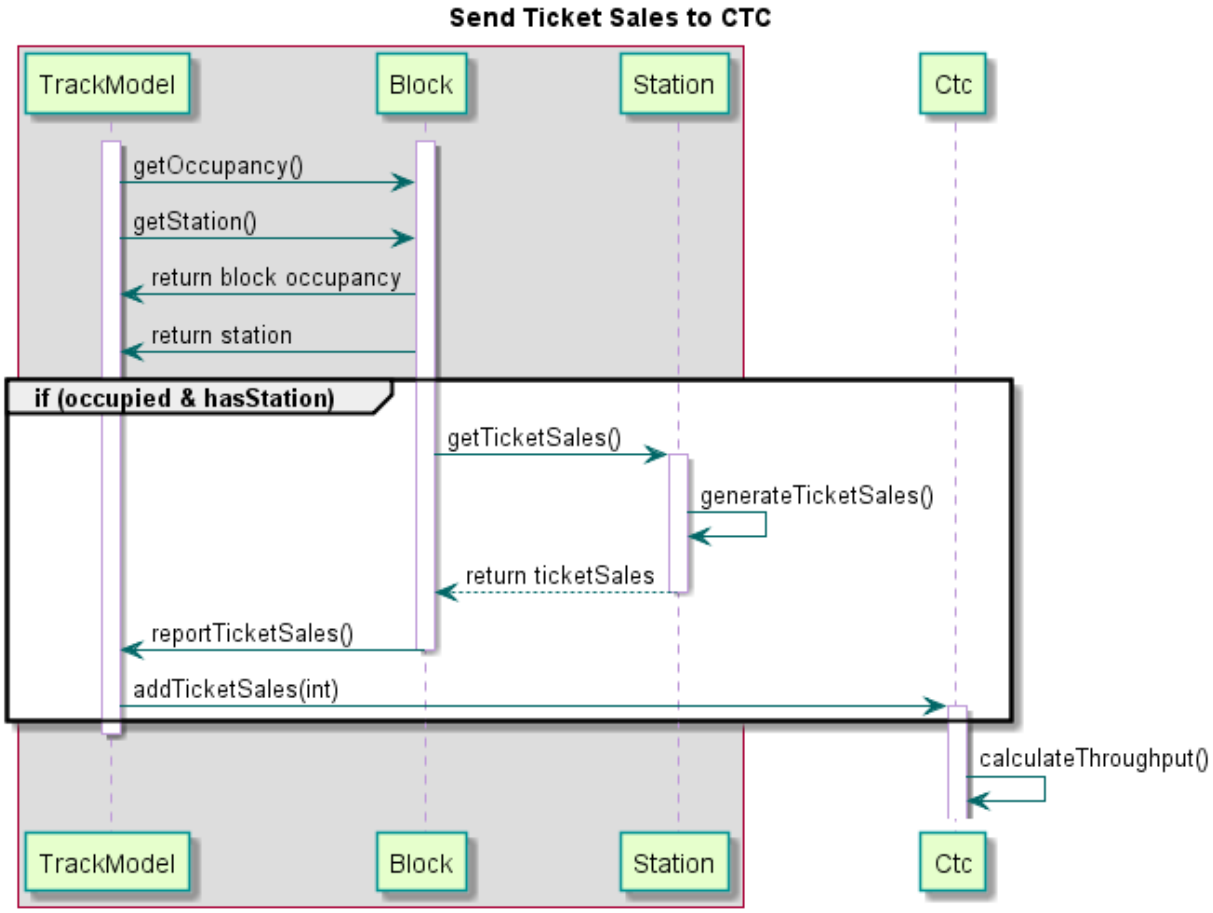
The following diagrams detail the data flow for the major use cases identified in Section 4.3.2, as well as the data flow for additional system-wide and internal functionalities.

4.3.5.1 Dispatch Train to Block



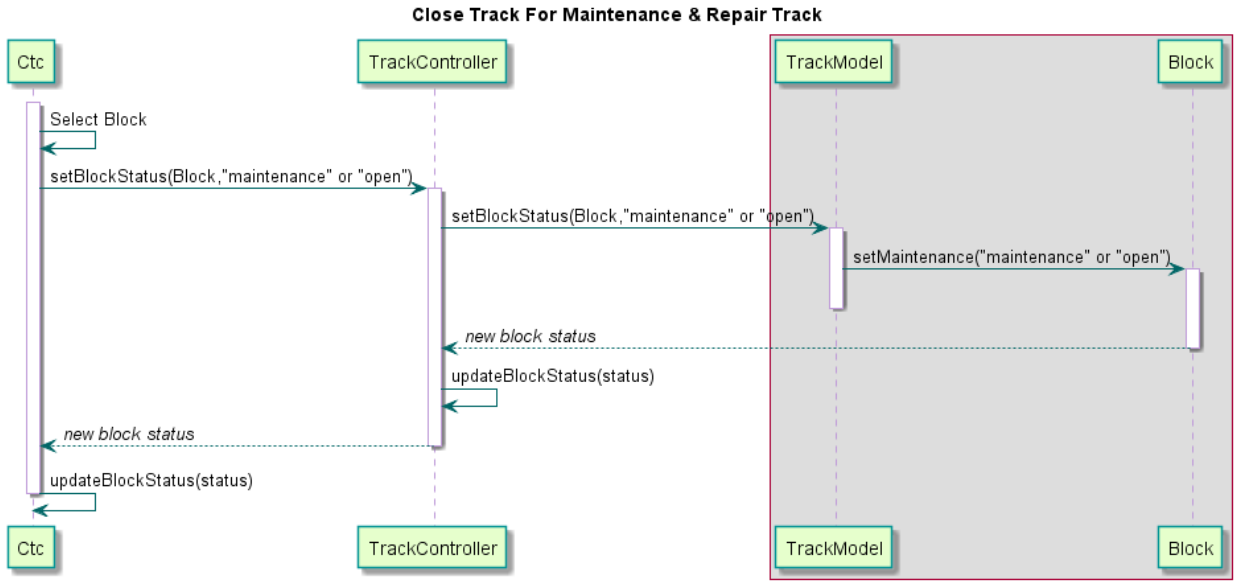
The diagram for the Track Model's role in dispatching a train to a block is shown above. When the CTC calls the Train Model module to instantiate a new train, the new train is dispatched with a starting block and schedule passed to the Train Controller module, which initializes and sends power to the train based on the block and speed limit of the dispatched location. At this point, the Track Model is used by the Train Controller to retrieve the speed limits for the line that the train was dispatched to upon initialization of the train.

4.3.5.2 Send Ticket Sales to CTC



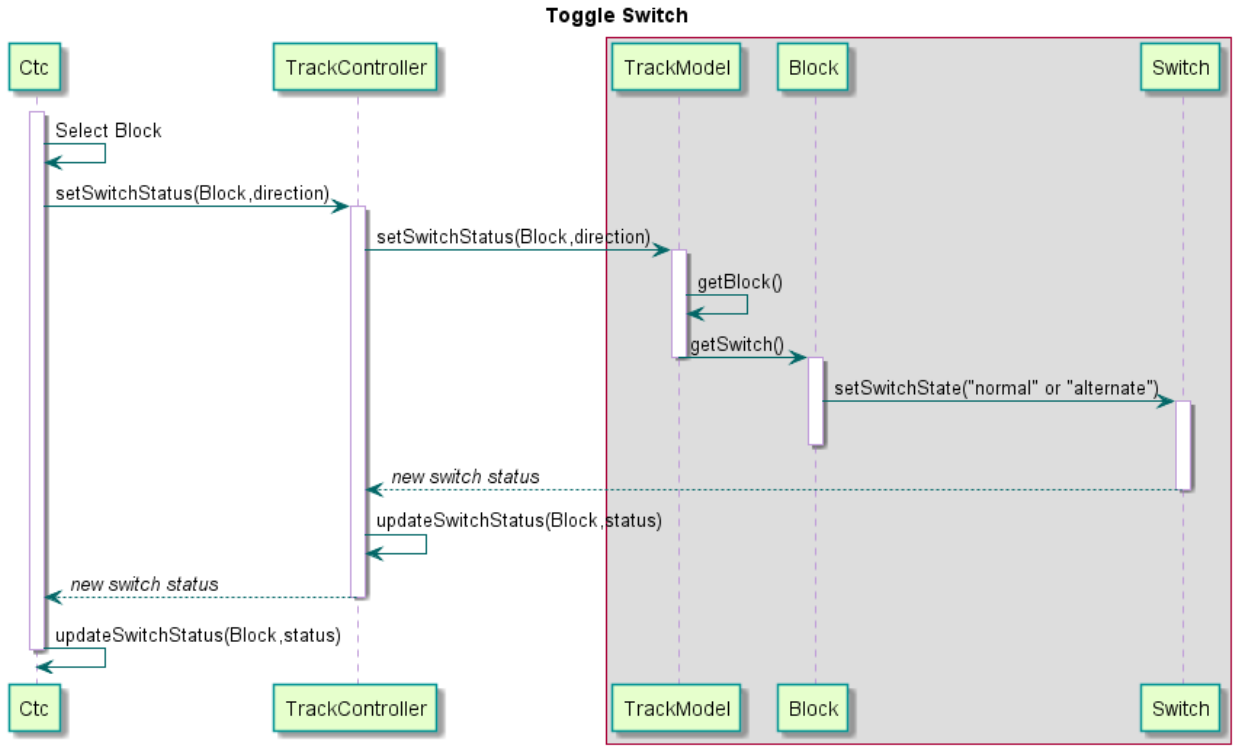
The diagram for the Track Model's role in reporting ticket sales to the CTC is shown above. When a block that contains a station is occupied by a train, a randomized number of ticket sales are generated from the Station object that the encompassing block reports to the overall Track Model. When the Track Model object is updated with ticket sales generated at a station, the Track Model then updates the CTC's ticket sales information in order for the CTC to use in its calculation of passenger throughput.

4.3.5.3 Close Track for Maintenance & Repair Track



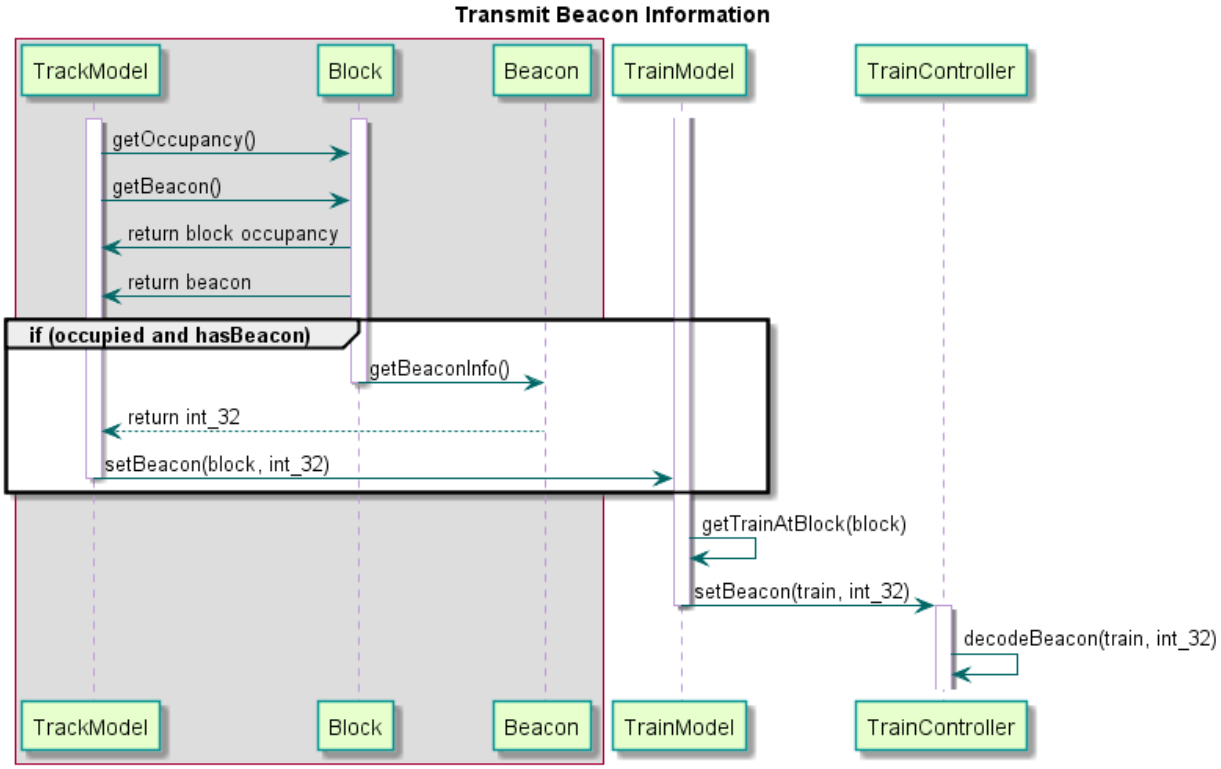
The diagram for the Track Model's role in closing a track for maintenance and repair is shown above. When the CTC transmits maintenance status information for a particular block to the Track Controller module, the new status is transmitted to the Track Model class which changes the maintenance status of the specified block to "maintenance" or "open". This new block status becomes visible to the Track Controller and CTC modules where the same path from CTC to Track Controller to Track Model may be invoked for toggling the maintenance status back to "open" upon track repair.

4.3.5.4 Toggle Switch



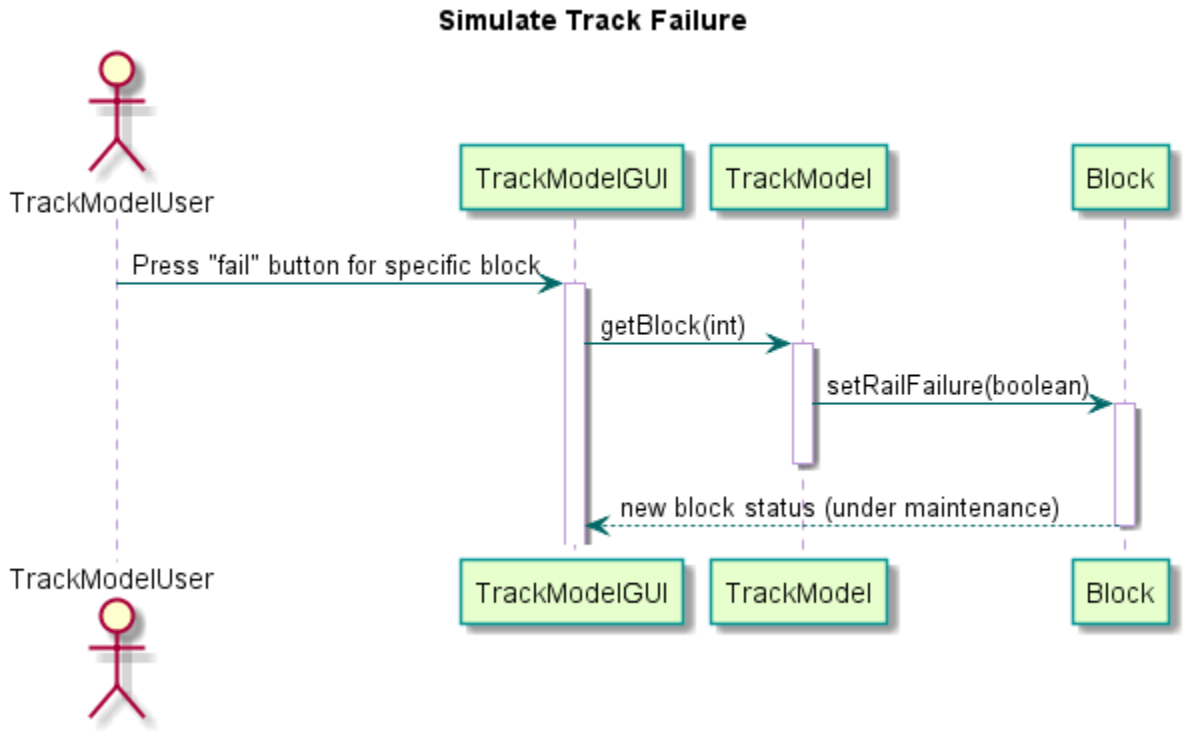
The diagram for the Track Model's role in toggling a switch is shown above. When a block that contains a station is occupied by a train, a randomized number of ticket sales are generated from the Station object that the encompassing block reports to the overall Track Model. When the Track Model object is updated with ticket sales generated at a station, the Track Model then updates the CTC's ticket sales information in order for the CTC to use in its calculation of passenger throughput.

4.3.5.5 Transmit Beacon Information



The diagram for the Track Model's role in transmitting beacon information is shown above. When a block that contains a beacon is occupied by a train, the 32-bit information associated with that beacon is set at the occupying train. Based on the GPS position of the occupying train determined by the Train Model class, the beacon information is then set at the Train Controller for the occupying train to be decoded and processed at the Train Controller level.

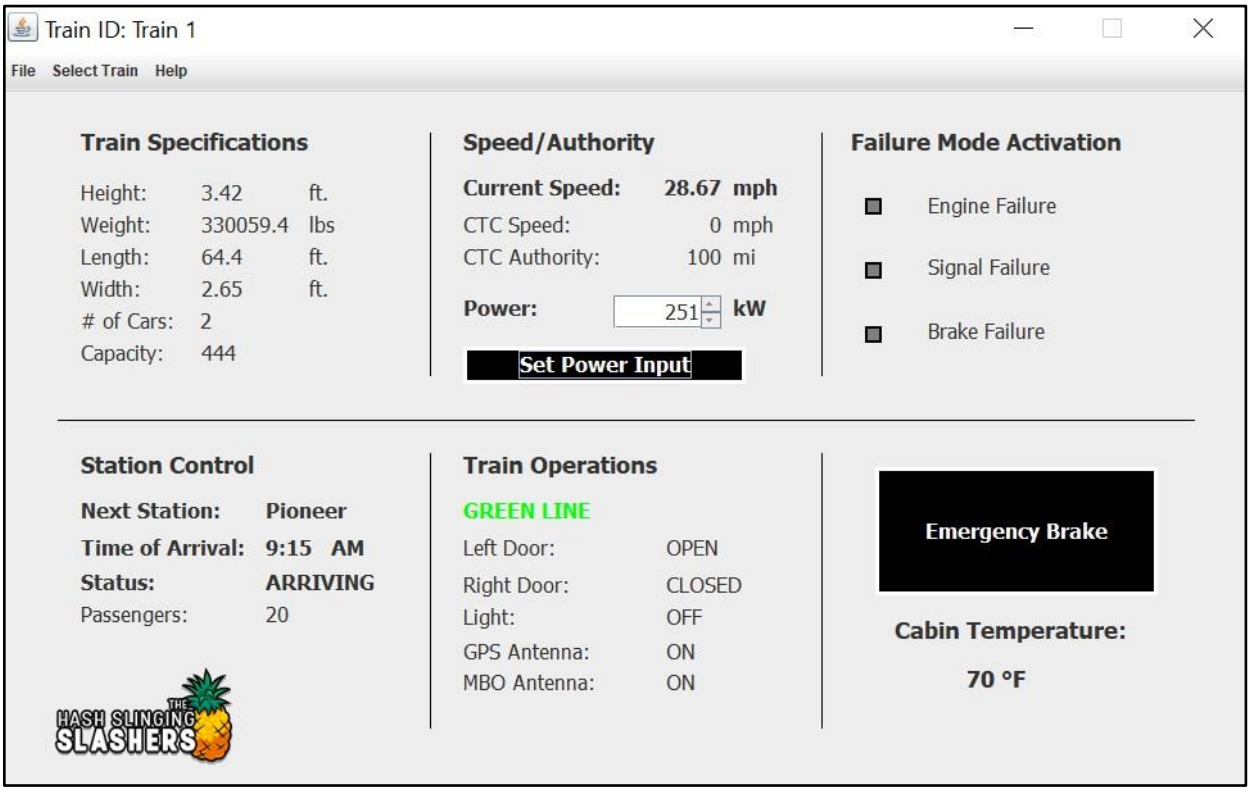
4.3.5.6 Simulate Track Failure



The diagram for the Track Model's role in simulating a track failure is shown above. Particularly, setting a rail to failure mode or broken status is depicted as one example track failure. When a Track Model user presses the "fail" button on the Track Model GUI with a specific block currently selected for view, that block will be accessed by the Track Model class and set to a failure status. This new status will place the block under maintenance and the status will be visible to the GUI user.

4.4 TRAIN MODEL

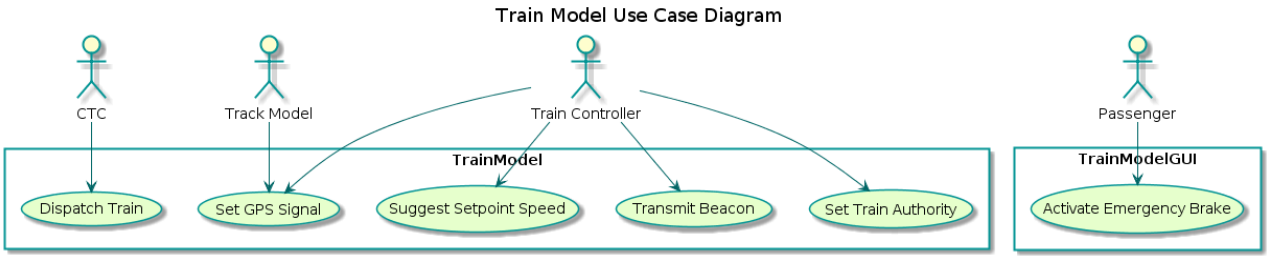
4.4.1 INTERFACE



The main screen of the Train Model displays all the inputs and outputs sent to/through the physical train whilst on certain section of a track. The Train Model GUI displays any relevant information, physical characteristics, inputs and outputs associated to the train itself. Each section of the GUI is divided into related information grouped in an intuitive manner. The critical components to note are the power input and current speed displays under the Speed and Authority section. As demonstrated by the Subsystem UI Presentation, the Train Model's key feature is its ability to take a power input from the Train Controller and determine the its desired velocity based on that power.

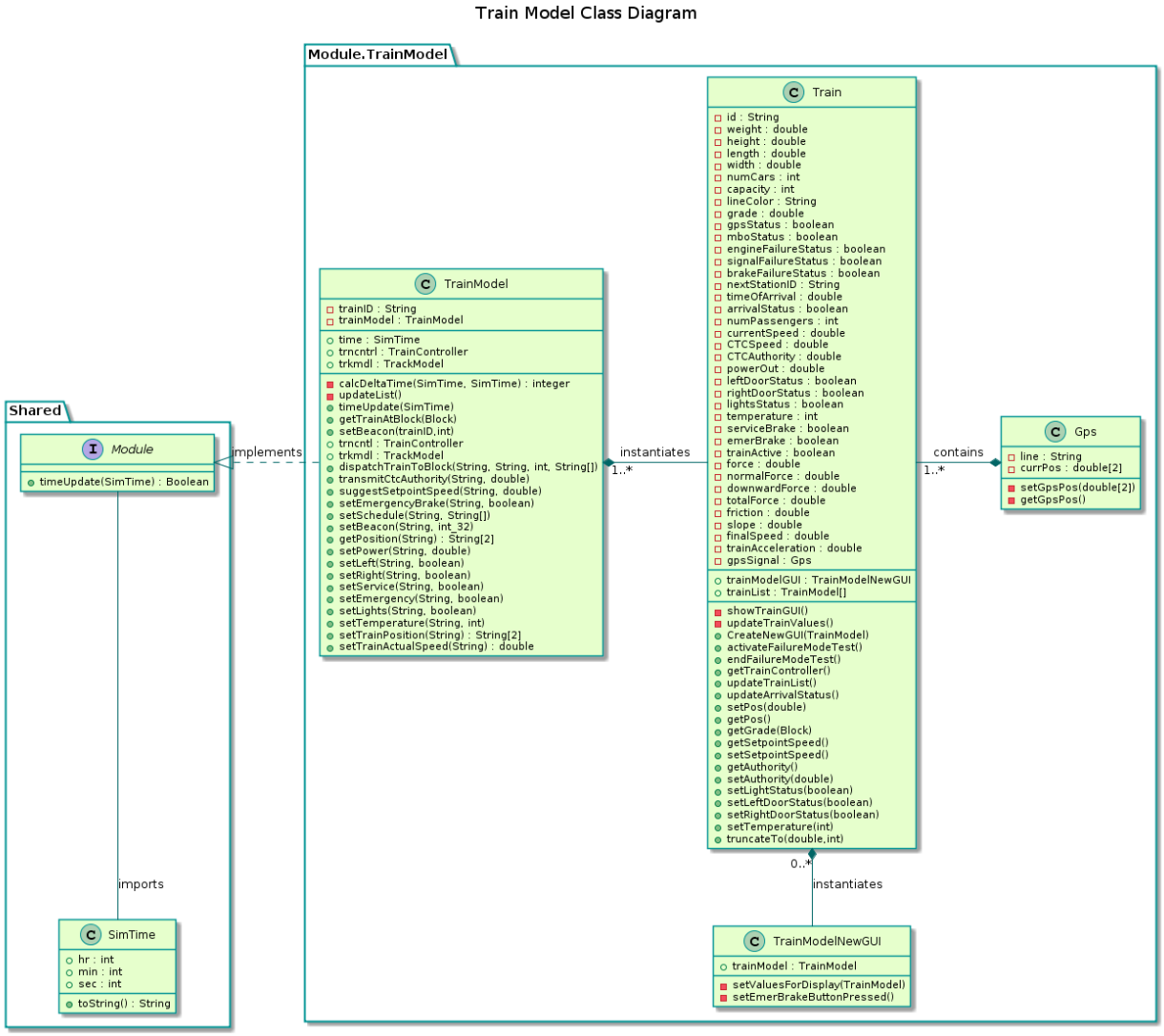
4.4.2 USE CASE

The Train Model as depicted in the following diagram implements seven primary use cases for a variety of users spanning passengers of the train to individual modules to assisting in the retrieval and transmitting of crucial data for system implementation. **The execution flow of the use cases depicted in the following diagram are explicitly diagrammed and detailed in section 4.4.5 of this document.**



4.4.3 CLASS DIAGRAM

The Train class serves to perform the primary operations of the Train Model (as defined in the SRS) including retrieving/transmitting CTC commanded speed and authority from the Track Model to the Train Controller, respectively.



The class diagram for the Train Model is shown above. The main class within the Module.TrainModel package is TrainModel. This class implements Module, connects to other modules, instantiates a Train object per active train at the time of it's dispatching by the CTC which in turn implements a GPS class to record information about its

position at any given time, and instantiates a Train Model GUI whenever the user selects a train model GUI to view, and updates the position of a . Each of these main functionalities are described below:

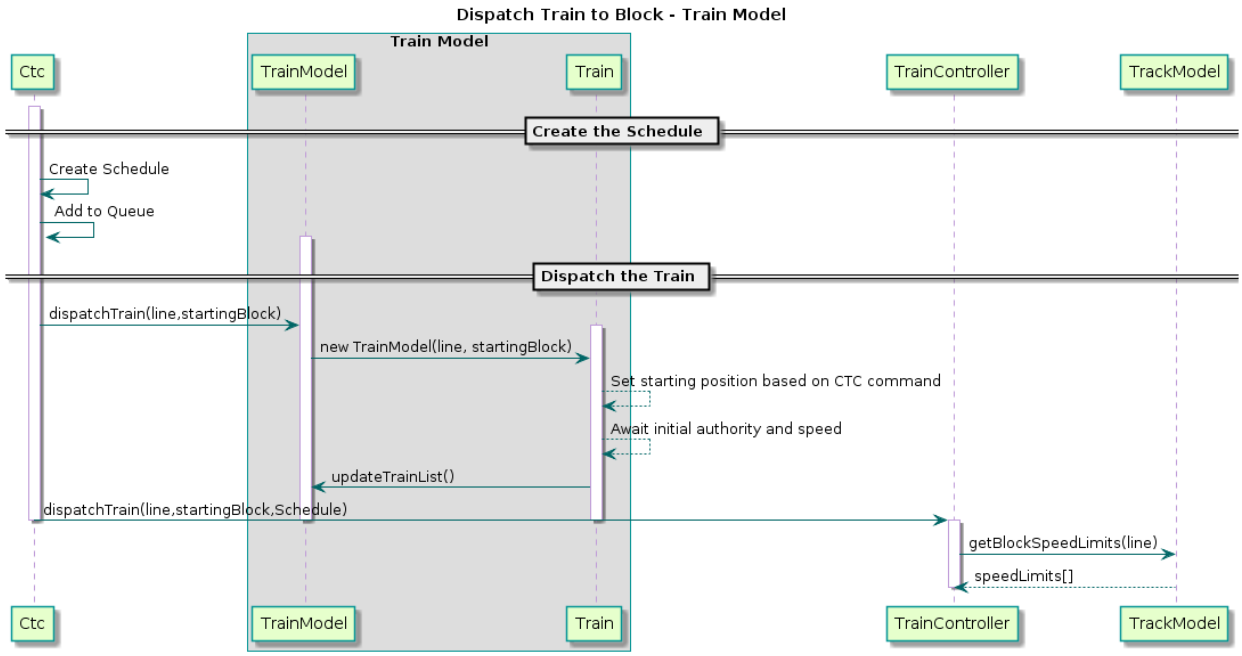
- Implements Module - Module is implemented (just as in every other North Shore Extension module) to contractually require the *timeUpdate(SimTime)* method within the Train Model. As described in Section 3, this function is meant to serve as a "rising clock edge" signalling the passage of one virtual second to the module. This function will be used within the Train Model to repaint any open train model UI's and update the status, attributes, and outputs of the train at a given time.
- Connects to other modules - The Train Model class has references to the Track Model and Train Controller main classes. These cross-references will be set by the Simulator immediately after all modules are initialized, as shown in the Simulator state diagram in Section 3. The reasonings for providing direct communication routes to these modules are as such:
    - Simulator - This path will be used to start/pause, as well as control the simulation speed from within the Train Model class. These functionalities have yet to be included within the GUI shown in Section 4.4.1.
    - Track Model - This path will be used to communicate track characteristics and track status from the Track Model through the Train Model to the Train Controller, allowing the controller/driver to consider physical factors for train movement as well as allowable directions of travel.
    - Train Controller - This path will be used for the Train Controller to retrieve the setpoint speed and authority suggested by the CTC, set a power command based on the suggested speed, update characteristics of the train based on its location on the track (i.e. lights, doors, etc.). The Train Model will also use this path as a means of communicating the x and y coordinates of its current position to the MBO.
- Instantiates the Train Object - The TrainModel class will serve as the liason between other modules and individual Train objects, as it contains a HashMap of train objects indexed by a hash code of their respective ID's. Any changes to an individual train object from an "outside" source will have to first go through this overarching class. The Train object will have a reference back to the TrainModel class to communicate with the other modules on a per-train basis.
    - Gps - Each Train object utilizes a Gps class to determine its current position on the track at a given time during the simulation. This information can then be passed on to other modules who call upon the Train Model class to receive this information.
- Instantiates the Train Model GUI - The Train Model will create a start up screen on initialization by the simulator which will serve as an interface allowing the user to select an initial train model to view and interact with. The TrainModelNewGUI will have a reference back to the TrainModel to communicate with the system and receive updates to reflect on the GUI.

4.4.4 DATA DESIGN

Data Being Stored	Proposed Data Structure	Rationale
List of Train objects, where the Trains within the list are active (have been dispatched to the track).	HashMap<String,Train>	The TrainModel class will implement a HashMap data structure of type String for retrieving a specific Train object by its string ID set at the instance of dispatching (handled by the CTC). This HashMap will be accessible to other modules through the overarching Module class implemented by all modules in the system.

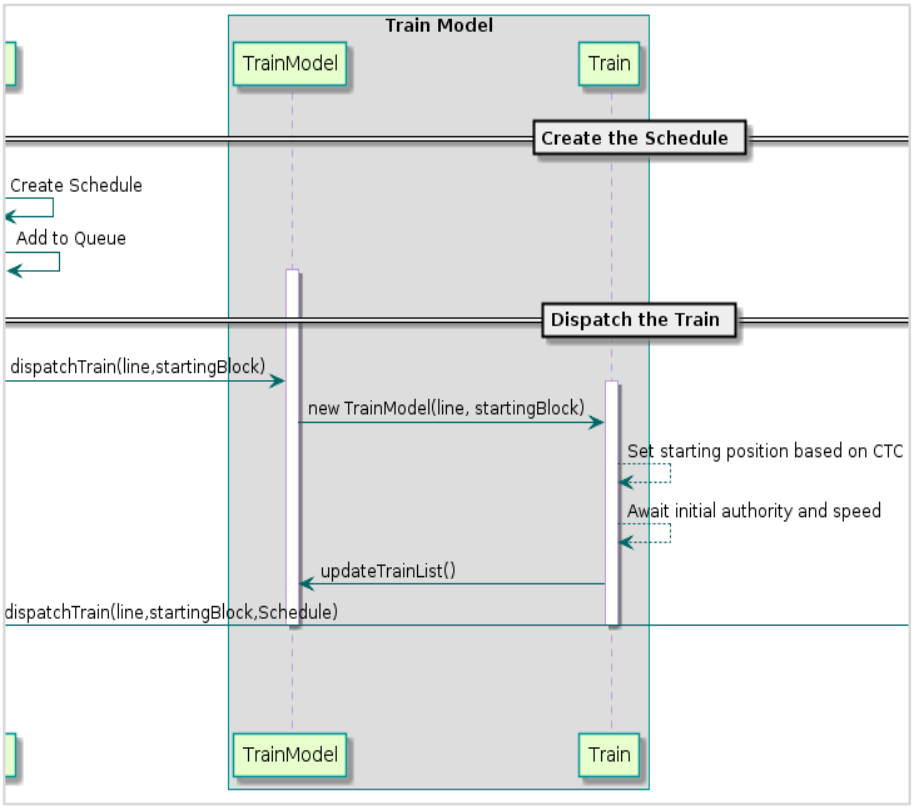
4.4.5 SEQUENCE DIAGRAMS

4.4.5.1 Dispatch Train - Train Model

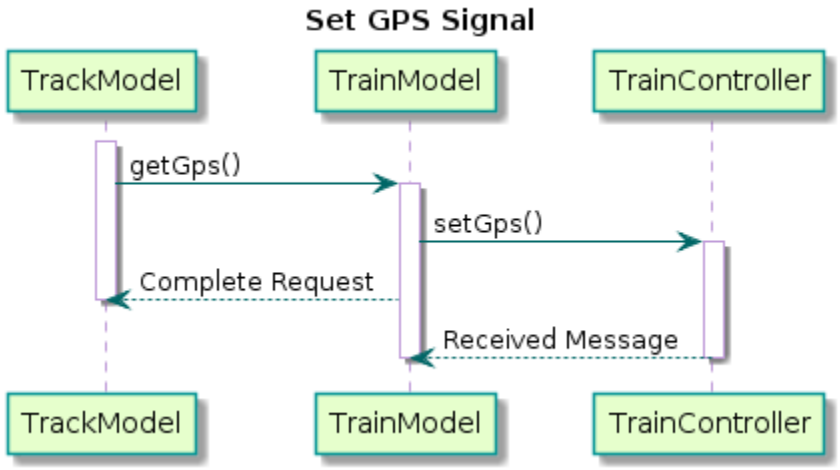


Dispatching a train is one of the key features of the system's design and is crucial for the implementation of the Train Model. Train objects cannot be instantiated without first receiving a signal from the CTC, who provides a train object the necessary

information to “exist” as an active train to move around the track with a speed and authority. The TrainModel class will receive the line of transport, a block ID and a train ID from the CTC. The TrainModel class will then instantiate a new Train object and add the object to the list. Below, there is a zoomed view of the Train Model’s responsibilities upon dispatching.



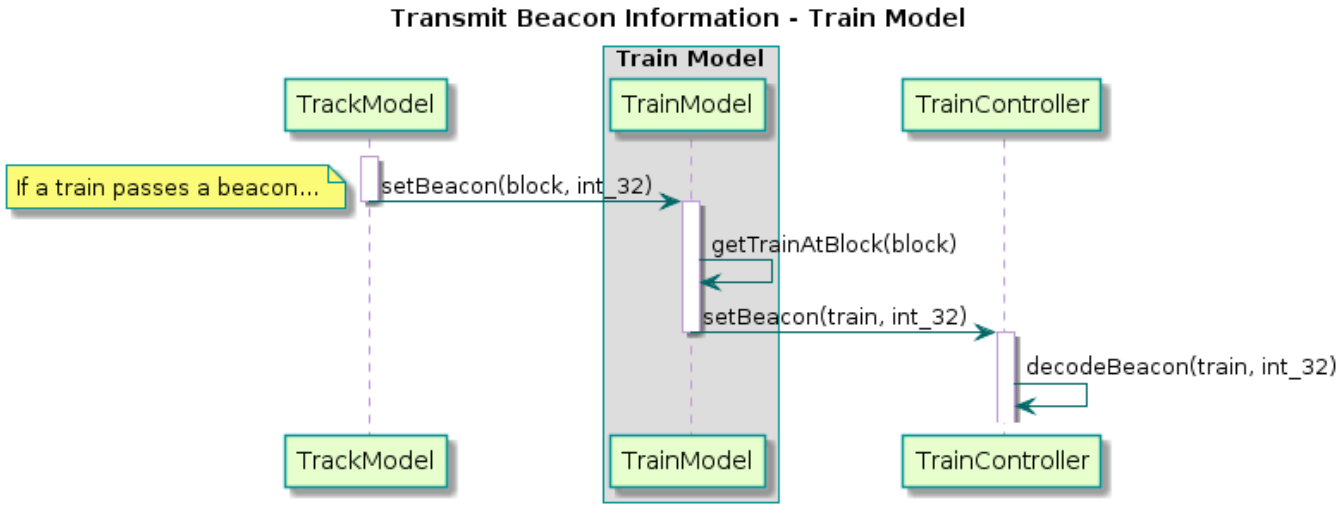
4.4.5.2 Set GPS Signal



The Train Model is responsible for the implementation, maintenance and execution of the GPS signal utilized by the system to keep record of the train’s position on the track

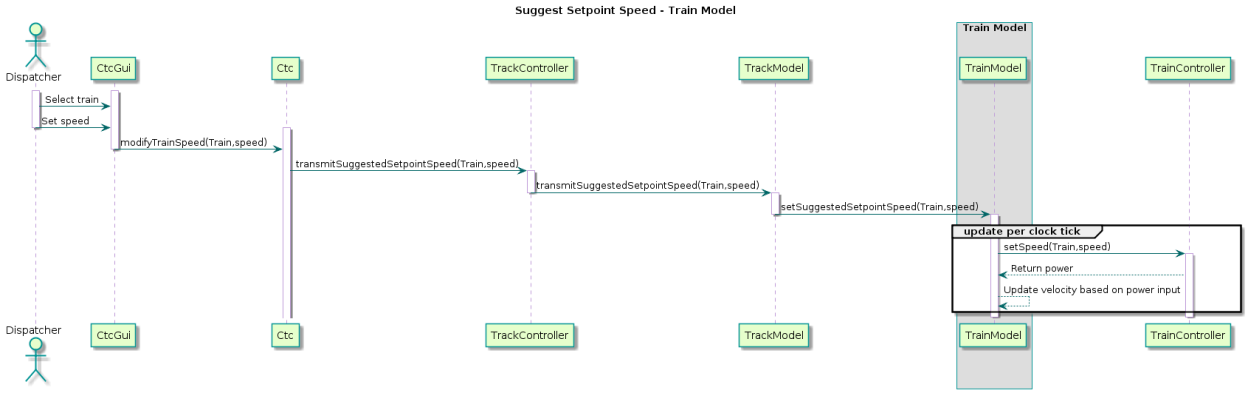
at any given moment. This signal is transmitted to the Track Model and the Train Controller as both an x and y position.

4.4.5.3 Transmit Beacon Information - Train Model

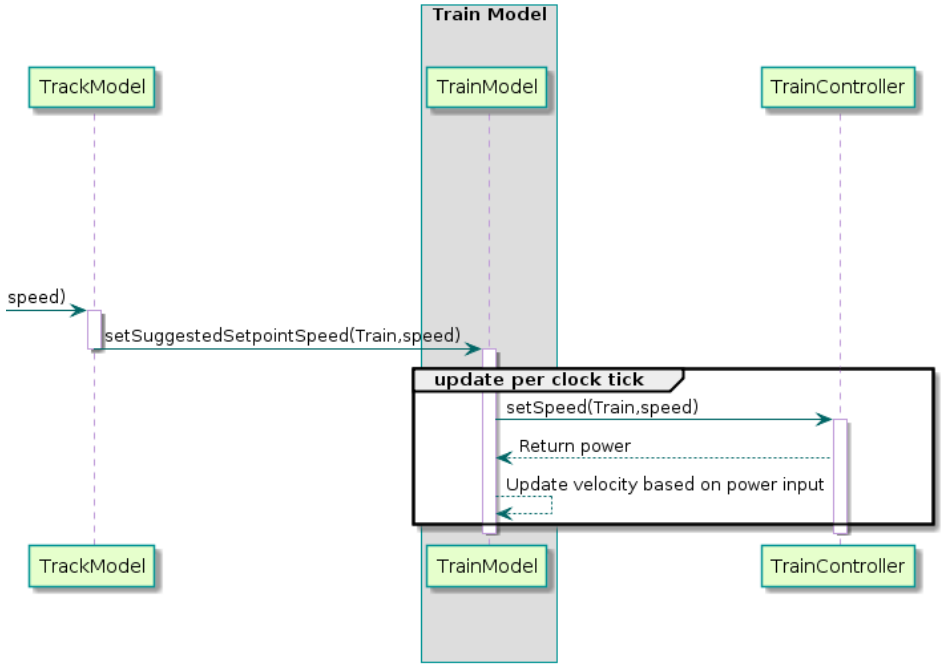


The Train Model assists the Track Model in passing along beacon information to Train Controller when a train passes a beacon a specific block to indicate arrival and departure from a station.

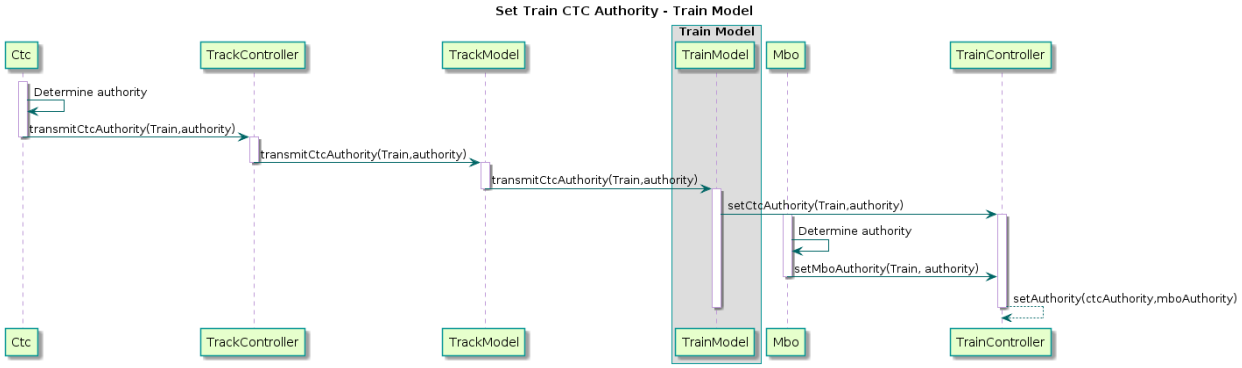
4.4.5.4 Suggest Setpoint Speed - Train Model



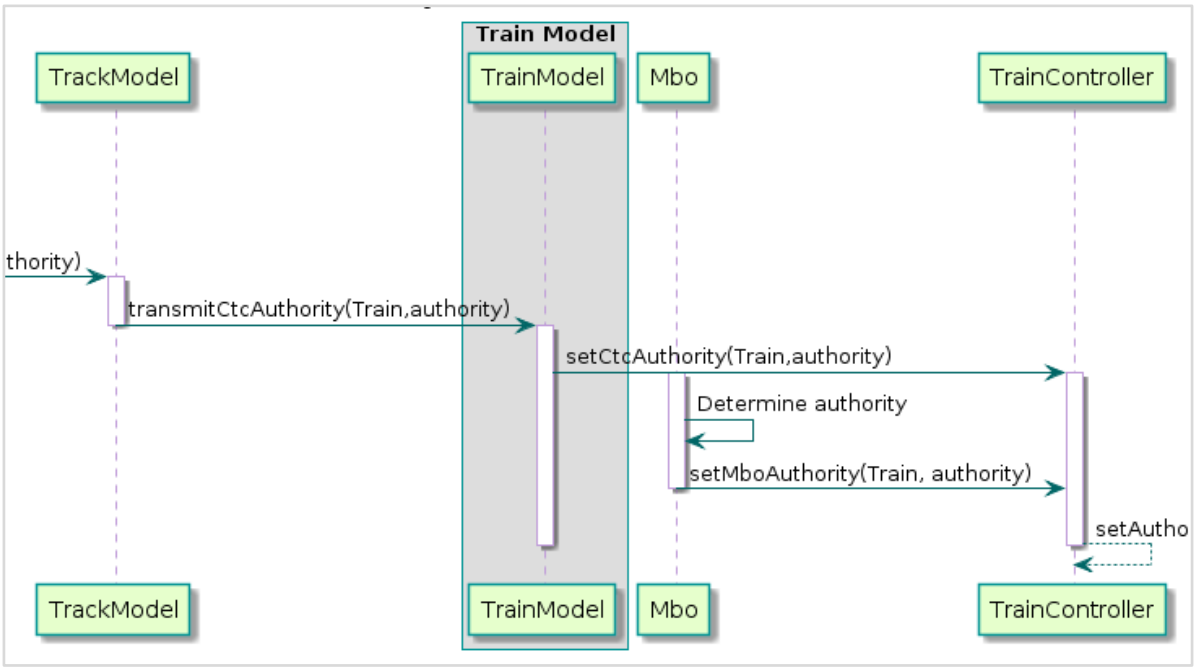
The Train Model is responsible for the retrieval and transmission the CTC commanded inputs for speed and authority. The two users presented in this use case are the Train Model's neighboring submodules, i.e. the Track Model and Train Controller. The Train Model receives the signal from the Track Model and passes it along to the Train Controller. Below there is a zoomed view of the Train Model's responsibilities during the transmission of a setpoint speed.



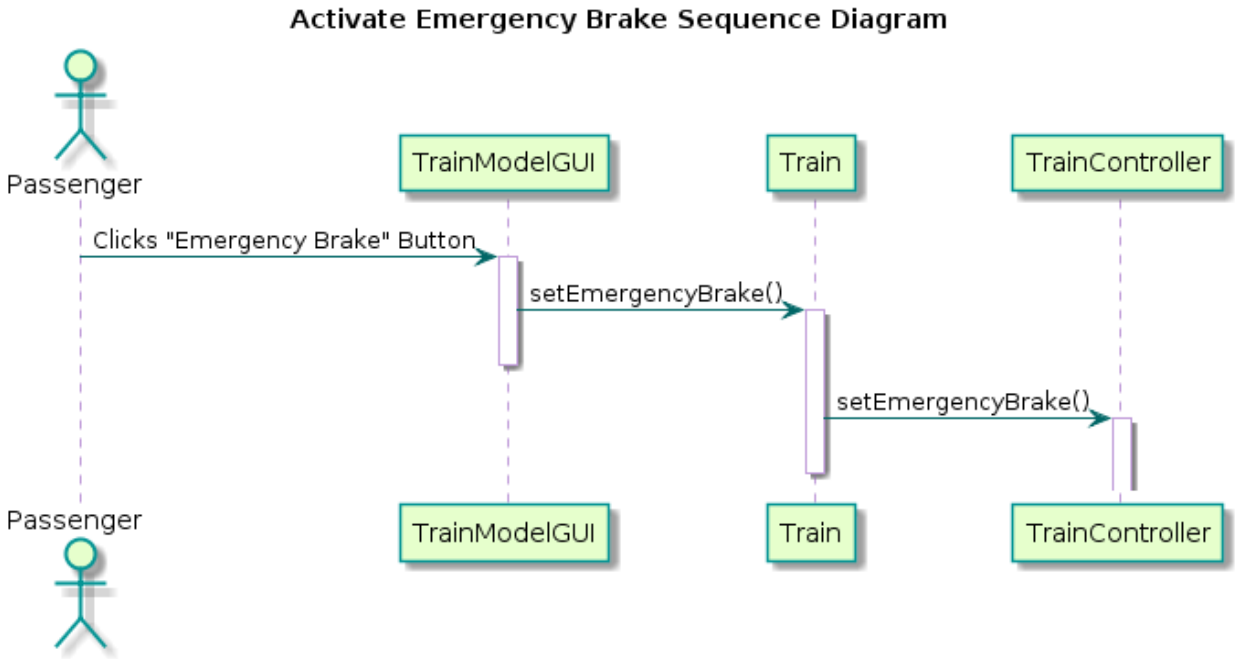
4.4.5.5 Set Train Authority - Train Model



As with setpoint speed, the Train Model has the responsibility to retrieve a suggested setpoint speed that originates as CTC input and travels, eventually to the Track Model submodule where the Train Model accepts the information. The Train Model is then responsible for transmitting this information onto the Train Controller submodule.



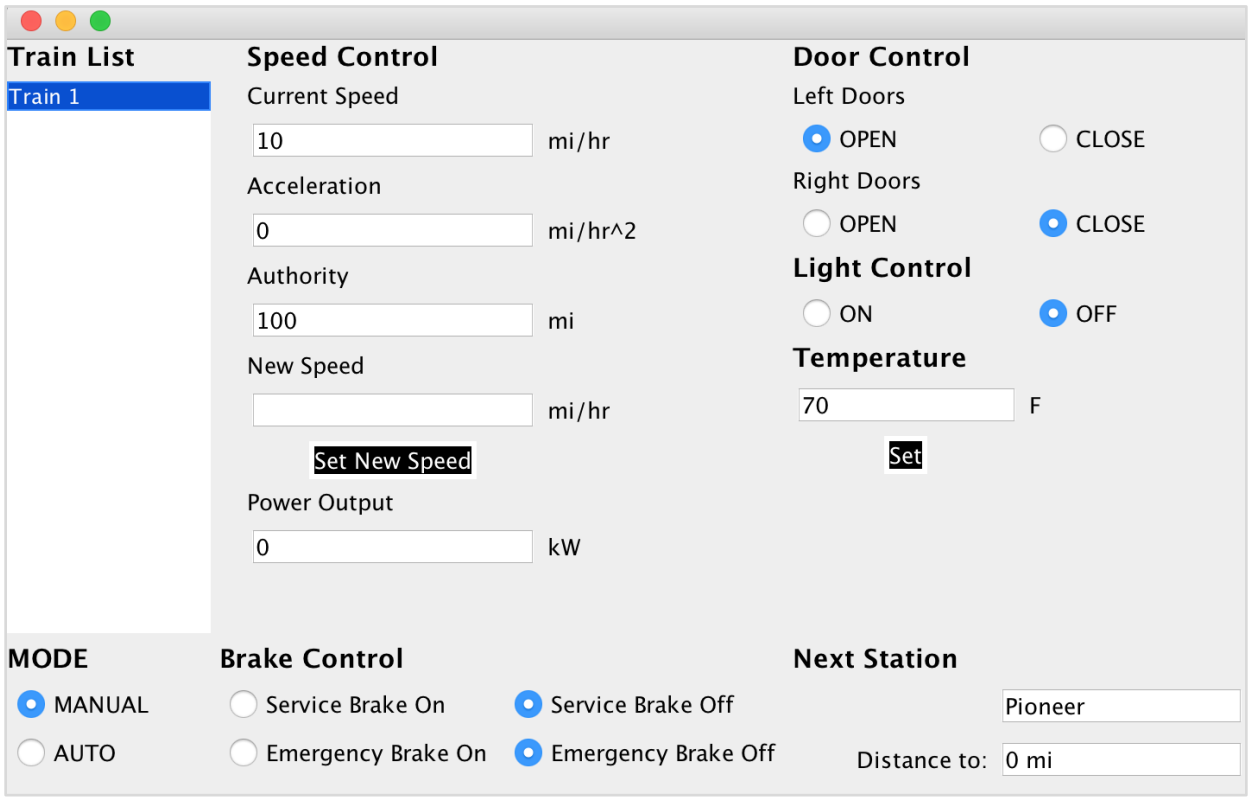
4.4.5.6 Activate Emergency Brake



The defined user of the Train Model's emergency brake functionality is the Passenger(s) of the train during the instance of emergency brake activation. As part of the requirements specified by the customer, the Train Model provides passengers the ability to pull the emergency brake to bring the train to a complete stop without permission from the Driver.

4.5 TRAIN CONTROLLER

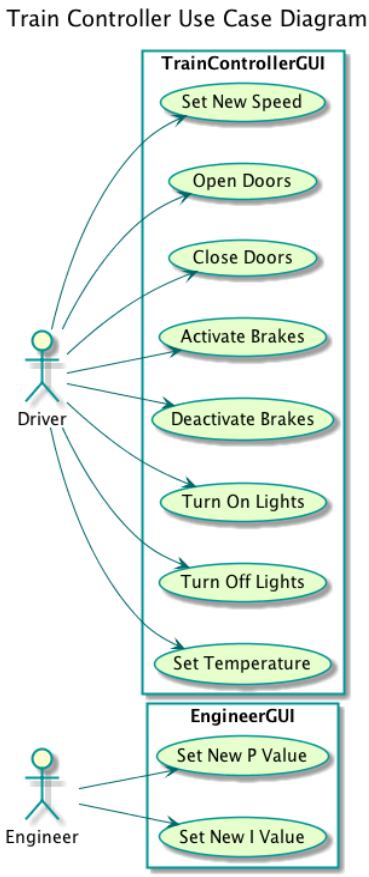
4.5.1 INTERFACE



This is the preliminary design of the graphical user interface (GUI) of the Train Controller. The interface design will change by the time the system is a finished product, but this preliminary interface shows all the important functionality of the GUI. In manual mode, the user of the interface, the driver, can enter a new speed for the train and press "Set New Speed" to change the setpoint of the currently selected train. The driver can enter a new temperature and press "Set" to change the internal temperature of the train's cabin. The driver can also change the statuses of the brakes, doors, and lights. Or the driver can switch the train into automatic mode, in which case all these functionalities would be performed autonomously based on information that the Train Controller receives from other modules.

4.5.2 USE CASE

The following diagram shows the actions that the two users of the Train Controller, the driver and the train engineer, can perform through the module. The driver can perform actions through the TrainControllerGUI and the engineer can perform actions through the EngineerGUI. These two interfaces have their use cases separated in the diagram. The use cases are explained in more detail below the diagram. The sequence flow for each of these use cases, as well as the flow for several Train Controller internal functionalities not controlled by either of these users, is detailed in Section 4.5.5 of this document.



The driver of the train will have the ability to enter a new speed in manual mode, which will change the setpoint speed of the train. This change of setpoint speed will then change the power output from the differential PI controller, which will be sent to the Train Model and change the actual speed of the train.

The driver of the train will have the ability to open or close the doors on either the left or right side of the train in manual mode. The new state of the doors will be sent to the Train Model, where the status of the doors will be modified.

The driver of the train will have the ability to activate or deactivate either the service or emergency brakes of the train in manual mode. The new state of the brakes will be sent to the Train Model, where the status of the brakes will be modified. Either the service or emergency brakes will then be activated, which will apply a negative acceleration and cause the actual speed of the train start to decrease.

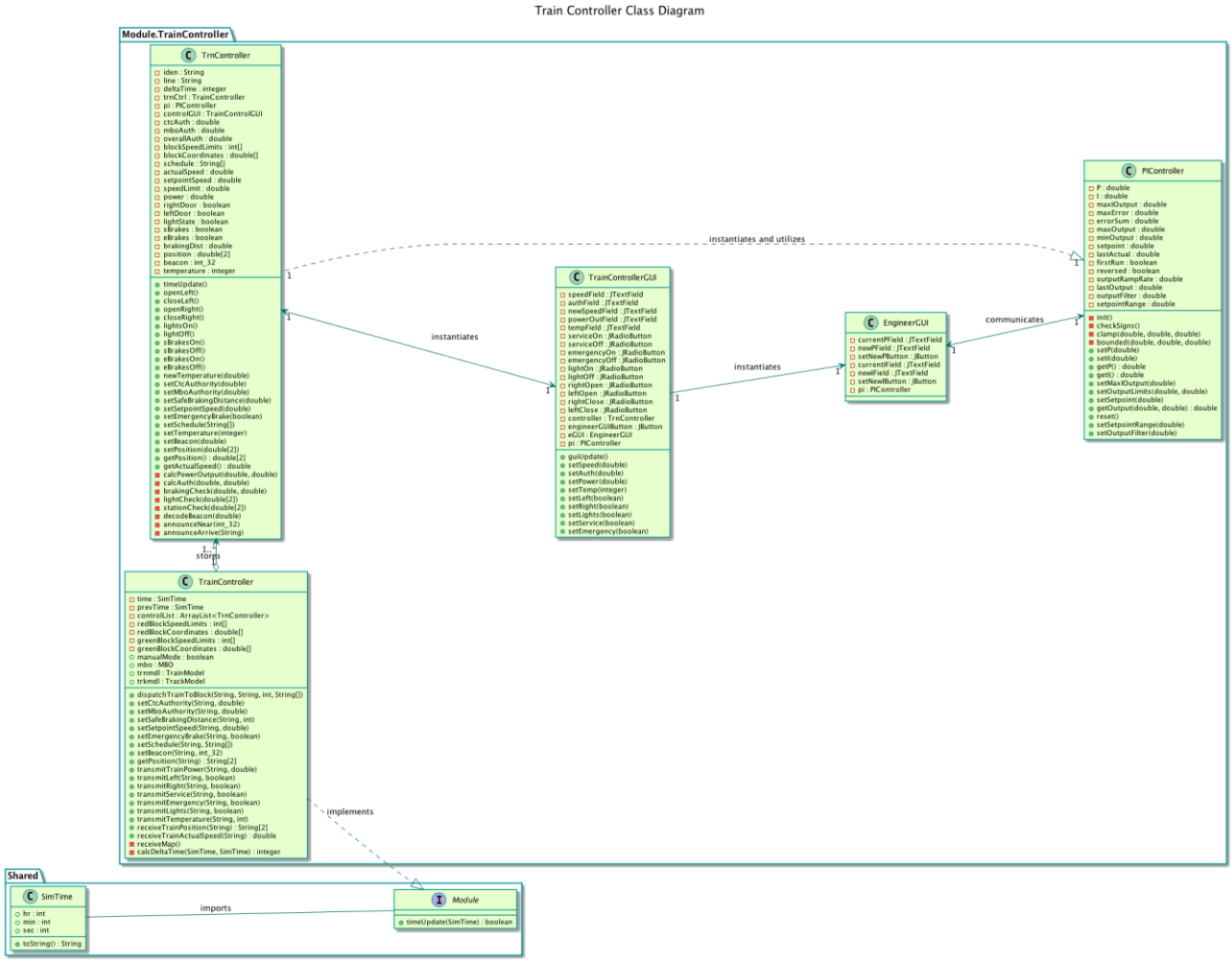
The driver of the train will have the ability to turn the lights of the train on or off in manual mode. The new state of the lights will be sent to the Train Model, where the status of the lights will be modified.

The driver of the train will have the ability to change the internal temperature of the train cabin in manual mode. The new value of the temperature will be sent to the Train Model, where the temperature of the train's cabin will be set to the value inputted by the driver.

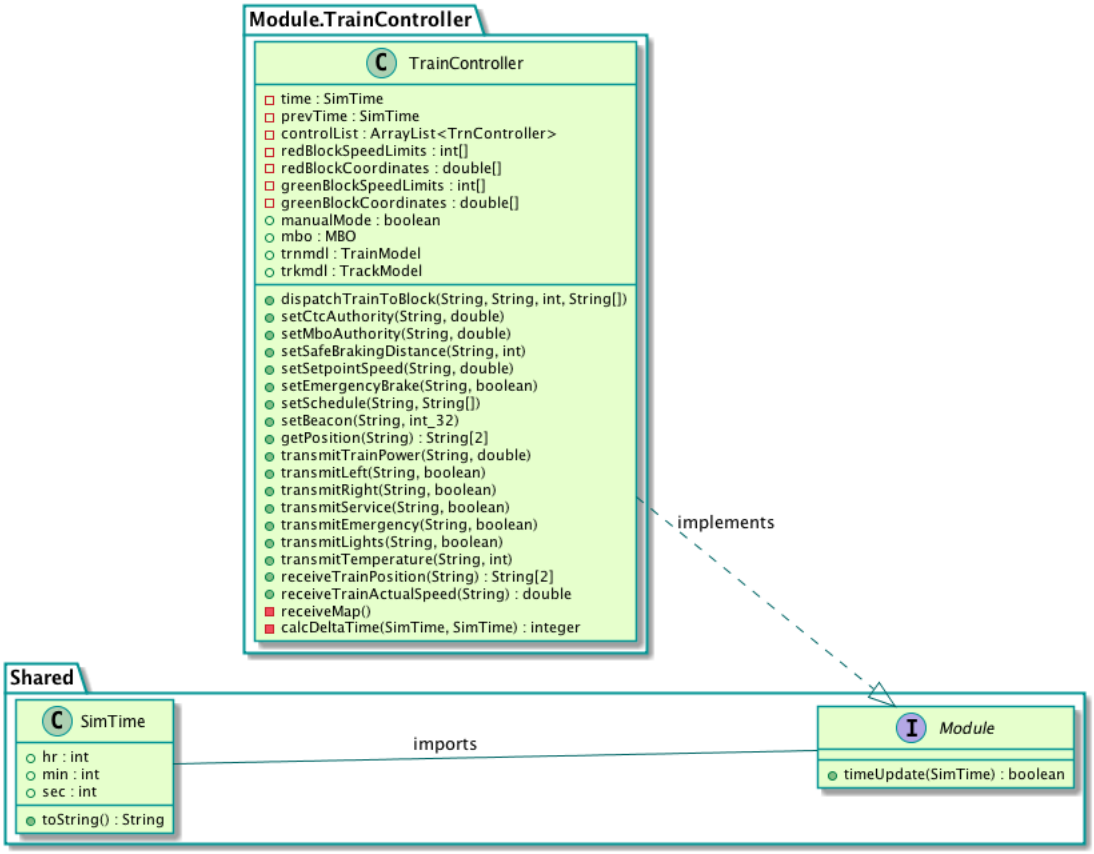
The engineer at the train yard will have the ability to change the K_P or K_I (referred to as P and I, respectively, for the remainder of this document and associated diagrams) value of the train's differential PI controller before the train departs. This will affect the power

level outputs that the controller calculates for the Train Model based on the setpoint speed and actual speed of the train.

4.5.3 CLASS DIAGRAM



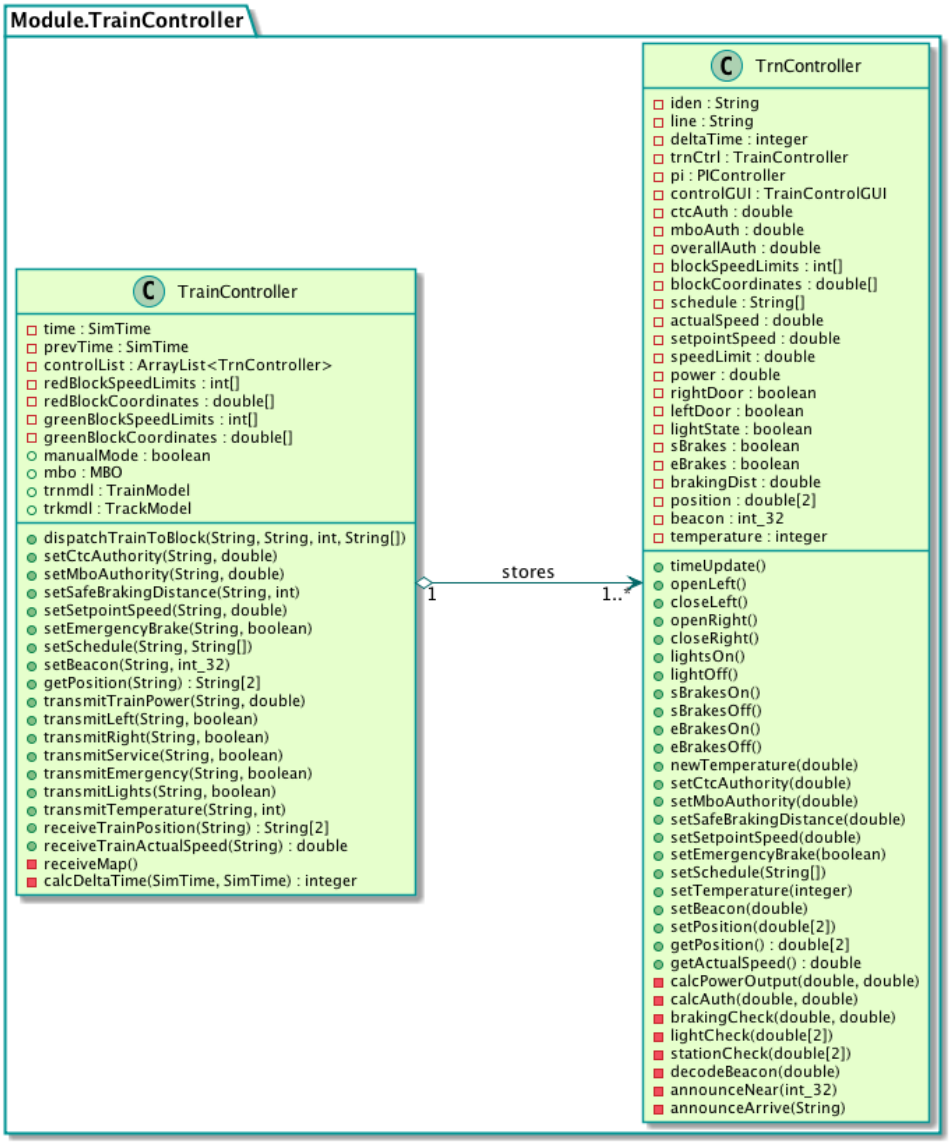
The full class diagram for the Train Controller is shown above. The main class within the Module.TrainController package is TrainController. This class implements Module, connects to other modules, and contains the aggregation of controllers for individual trains, which are described in the TrnController class. These functions of the TrainController class, including zoomed-in images of the relevant class diagram subsections, are further described below:



- Implements Module - Module is implemented (just as in every other North Shore Extension module) to contractually require the *timeUpdate(SimTime)* method within the Train Controller. As described in Section 3, this function is meant to serve as a “rising clock edge” signalling the passage of one virtual second to the module. This function will be used within the Train Controller to update the GUI for each train, calculate a new power for each train, determine a new position for each train, calculate a new authority for each train, receive a new safe stopping distance for each train, and determine if the brakes, doors, or lights of each train need to be activated or deactived. Implementing Module also imports SimTime which allows the Train Controller to determine the amount of time that has passed between each execution of *timeUpdate()*. This will normally be assumed to be one second, but that standard could theoretically be changed at some point in the usage of the system.
- Connects to other modules - The TrainController class has references to the Train Model, MBO, and Track Model. In addition, the CTC module will have a reference to the TrainController object for communication These cross-references will be set by the Simulator immediately after all modules are initialized, as shown in the Simulator state diagram in Section 3. The reasonings for providing direct communication routes to these modules are as such:
    - Train Model - This path of communication must be available for the Train Model to transmit position and actual speed to the Train Controller. The Train Model will need to set the status of the emergency brake in the Train

Controller if a passenger happens to trigger it. The Train Model will also need to send the Train Controller the value of any beacons it happens to pass on the track. On the flipside, the Train Controller will need to communicate with the Train Model in order to transmit the power level of the engine and the status of the brakes to affect the speed of the train. The Train Controller will also need to transmit the status of the doors and lights, the internal temperature value of the train's cabin, and a signal to notify the Train Model that the train is either nearing or arriving at a station.

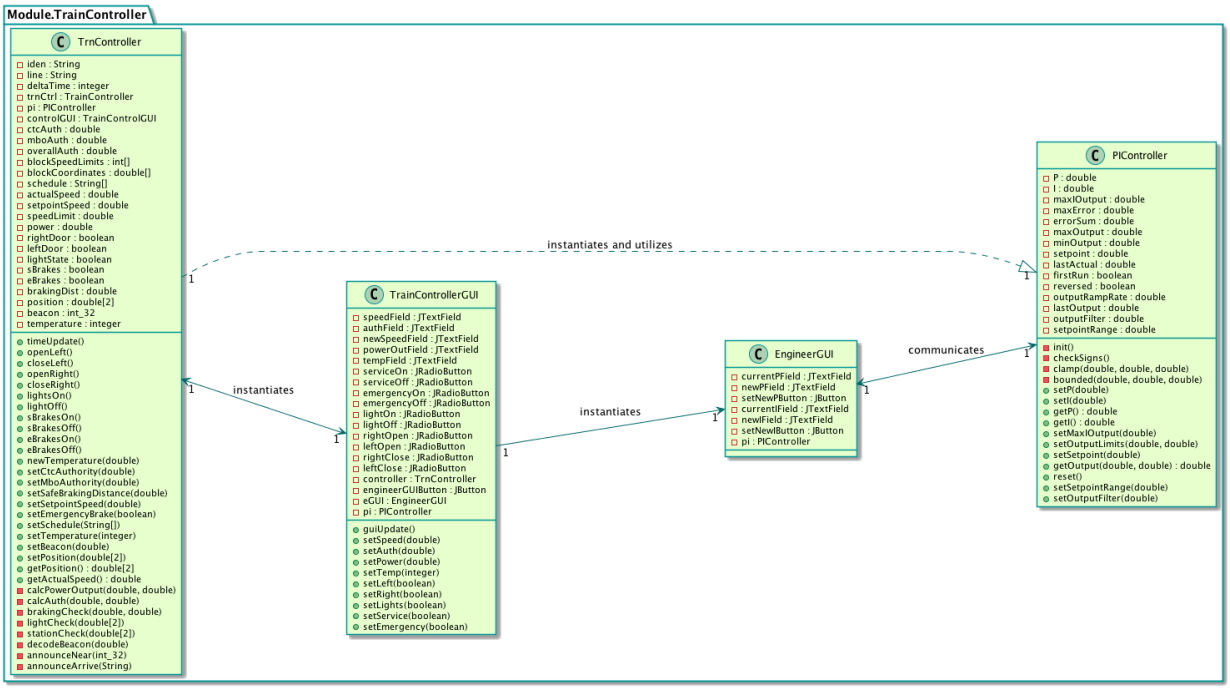
- MBO - This path of communication must be available in order for the MBO to get the position of each train from the Train Controller. The MBO will also send each train's controller an authority and a safe stopping distance to ensure that the train stops safely without any collisions occurring.
- Track Model - This communication path will be used solely for fetching the lists of coordinates and speed limits for each line of the track. These lists will be fetched on initialization of the system and will be stored in the TrainController object. When each new TrnController object is instantiated, the TrainController object will give it the lists of the coordinates and speed limits of the line it will be operating on.
- CTC - The Train Controller module will not have a reference to the CTC module, but the CTC will have a reference to the Train Controller. This communication path will be used solely to notify the TrainController object that a new train has been dispatched, and to give it any necessary information to initialize a new TrnController object.



- TrnController aggregation - The TrnController class is responsible for storing information and functions for each individual train. The TrainController class serves as an overall object, with its most important feature being the ArrayList<TrnController> that it stores. A new TrnController object will be instantiated into this ArrayList whenever the CTC dispatches a new train onto the track. There is only one TrnController object instantiated per active train on the track.

The other major class in the Train Controller module is the TrnController class. This class is responsible for storing the values and controlling the functions of each individual train. The TrnController object instantiates and utilizes a PIController object to calculate a power level for the train's engine. Each TrnController object also instantiates a new TrainControllerGUI object for the train. Each of these TrnController class functions, including zoomed-in images of relevant class diagram subsections, are explained in further detail below:

- Controlling the train - The TrnController object is responsible for storing relevant information and executing functions pertinent to the correct and safe operation of each individual train. This includes calculating a new power, determining a new position, calculating a new authority, receiving a safe stopping distance, setting signals to announce that the train is nearing or arriving at a station, activating the brakes if necessary, determining if the train is in a station to open or close the doors, and determine if the train is underground to activate or deactivate the lights at every execution of the *timeUpdate()* function from the TrainController object.



- Instantiating a PIController object - The PIController class implements the differential PI controller responsible for generating a power output after being given a setpoint speed and the train's current actual speed. The PIController object is instantiated by the TrnController object of a train since TrnController is the class that utilizes the functionality of the PIController class. There is only one PIController object that is instantiated per train. However, a reference to the same PIController object will also be passed from the TrnController object to the EngineerGUI object to enable the train yard's engineer to modify the values of the P and I fields of the PIController object before the train departs.
- Instantiating a TrainControllerGUI object - The TrainControllerGUI class describes the user interface that will be used by the driver of the train. If the TrainController is in automatic mode, this GUI serves primarily as a data display. If the TrainController is in manual mode, the driver will provide many of the input values that contribute to controlling the train, including setpoint speed, boolean values for doors, brakes, and lights, and setting a temperature value. The TrainControllerGUI also instantiates an EngineerGUI object, through which the engineer at the train yard can modify the P and I values of the train's PIController object. These two functionalities are further explained below:

- Instantiating an EngineerGUI object - The TrainControllerGUI will instantiate an EngineerGUI object when a new train is created. This GUI will initially be hidden, and will appear when the "EngineerGUI" button is pressed on the TrainControllerGUI, which is not implemented on the preliminary GUI shown in Section 4.5.1, but will be included when it is completed. When the window is closed, the GUI will continue to run but will again be hidden from view.
- Setting new P and I values - The only purpose of the EngineerGUI will be to display the current values of P and I in the train's PIController object and allow these values to be changed by accessing get and set functions in the PIController object. The EngineerGUI will have a reference to the train's PIController object passed to it as part of its constructor function.

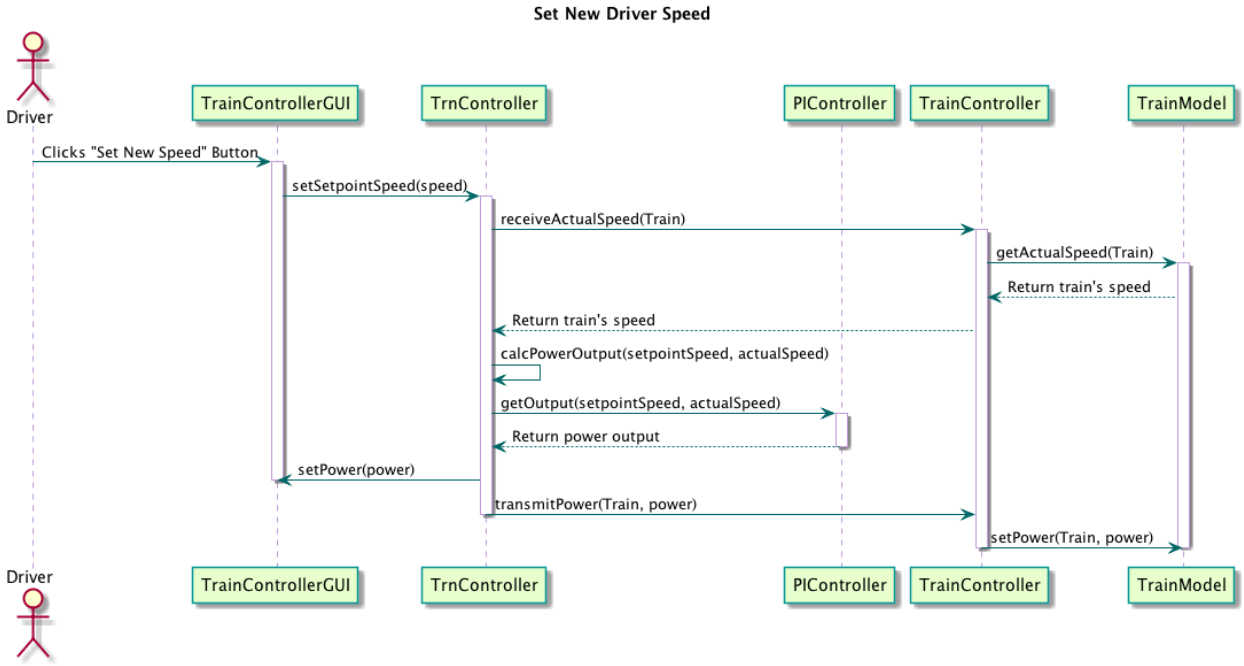
4.5.4 DATA DESIGN

Data Being Stored	Proposed Data Structure	Rationale
List of TrnController objects	ArrayList<TrnController>	It will be easy to search through the ArrayList for the TrnController object with a particular identifier String, and then perform operations on that specific object.
Position of the train	double[2]	The most logical implementation for storing an (x,y) coordinate is in a double array of size 2.
List of block coordinates and speed limits	double[2][] and int[], respectively	Knowing the coordinates from the Train Model, it will be simple to search the double matrix for the proper coordinates and find the corresponding speed limit from the same index in the integer array.
Schedule for a train	String[ ]	The simplest and most efficient way to store a schedule is to just store an array of station names that the train has to stop at.

4.5.5 SEQUENCE DIAGRAMS

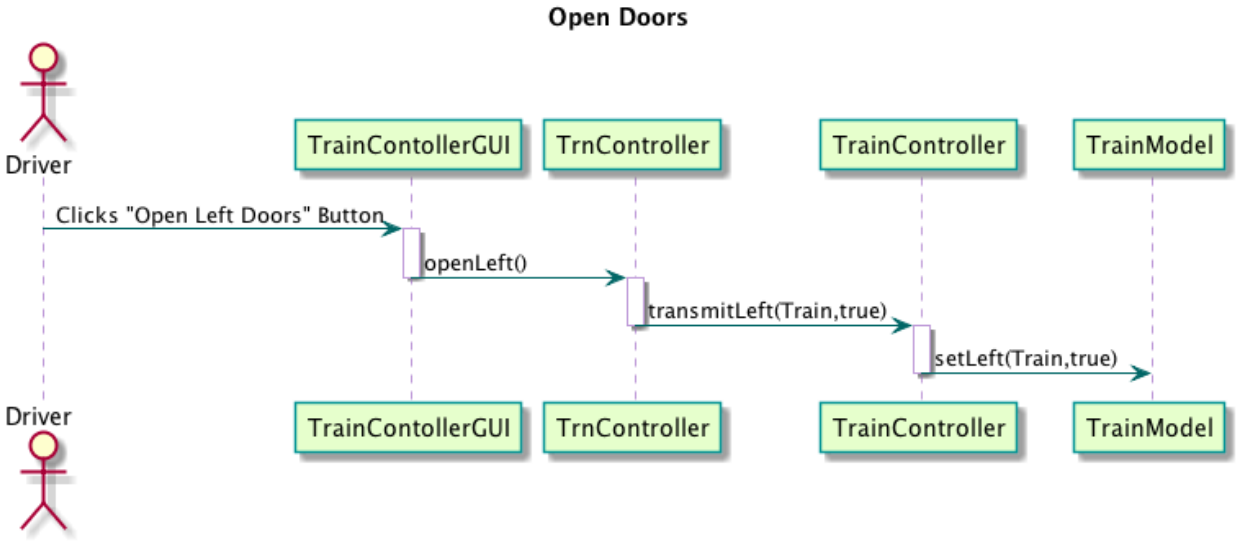
The following diagrams detail the data flow for each of the use cases identified in Section 4.5.2, as well as the data flow for additional system-wide and internal functionalities.

4.5.5.1 Set New Speed



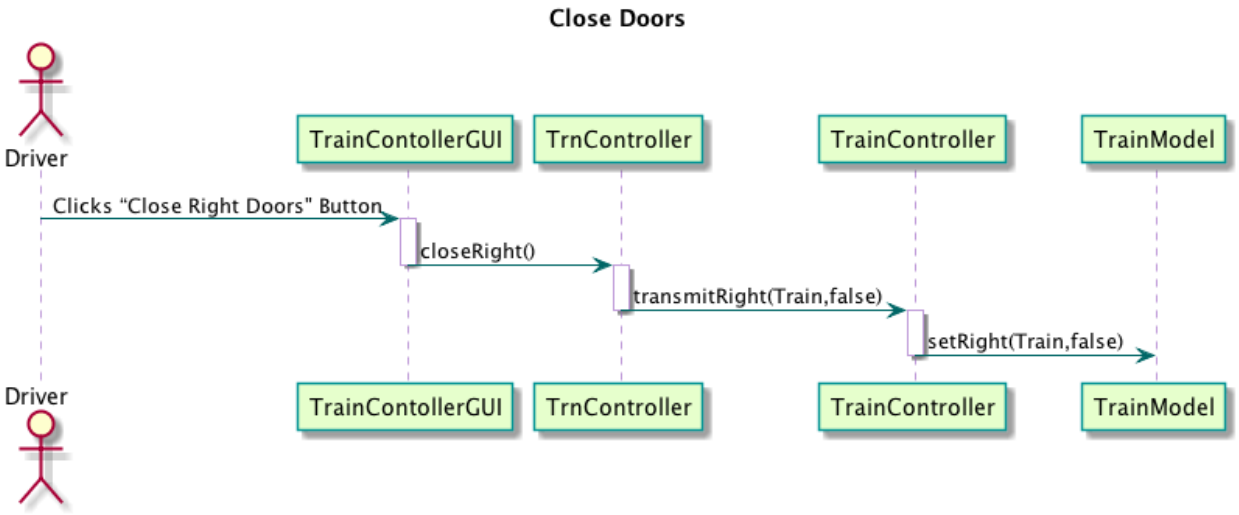
This sequence flow begins when the driver of the train enters a new speed and clicks the “Set New Speed” button on the TrainControllerGUI. The TrainControllerGUI object then sets the setpoint speed of its corresponding TrnController object to the driver’s input. Then the TrnController object requests the actual speed of its train, which the TrainController object gets from the TrainModel object using the train’s identifier string. With the setpoint speed and actual speed of the train, the TrnController calculates a new power output by getting the output from the PIController object. The TrnController then sets the power display of the TrainControllerGUI so the driver can see the new power value. Finally, the TrnController transmits the new power to the TrainModel through the TrainController, using the identifier string of the train to make sure the proper model object is modified.

4.5.5.2 Open Doors



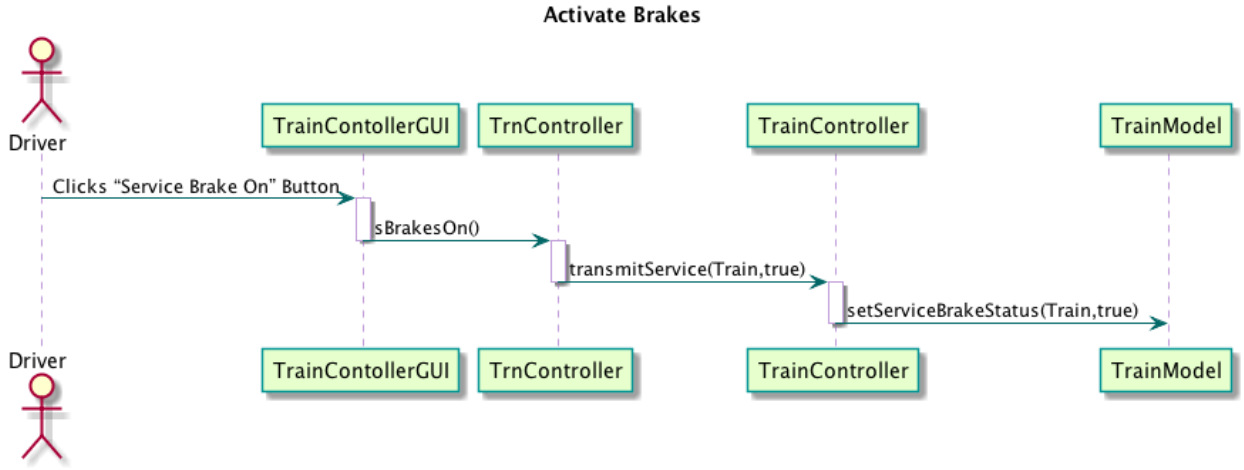
This sequence flow begins when the driver clicks on a button to open a set of doors on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController open the corresponding set of doors. Then the TrnController sends the new state of the doors, the boolean value “true”, to the TrainModel through the TrainController, using the train’s identifier string to ensure that the proper model object is modified.

4.5.5.3 Close Doors



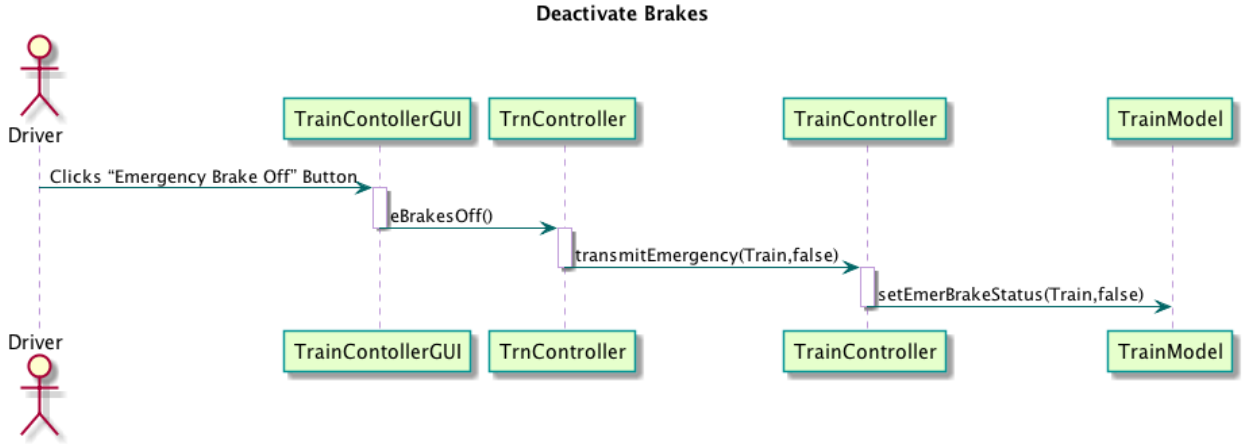
This sequence flow begins when the driver clicks on a button to close a set of doors on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController close the corresponding set of doors. Then the TrnController sends the new state of the doors, the boolean value “false”, to the TrainModel through the TrainController, using the train’s identifier string to ensure that the proper model object is modified.

4.5.5.4 Activate Brakes



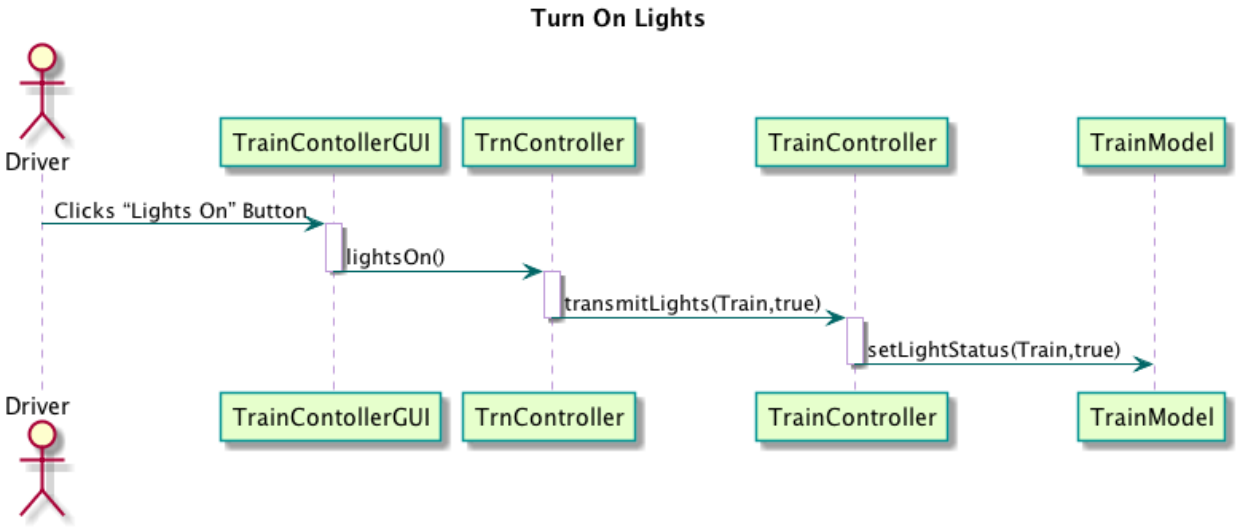
This sequence flow begins when the driver clicks on a button to activate a type of brakes on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController activate the proper brakes. Then the TrnController sends the new state of the brakes, the boolean value "true", to the TrainModel through the TrainController, using the train's identifier string to ensure that the proper model object is modified.

4.5.5.5 Deactivate Brakes



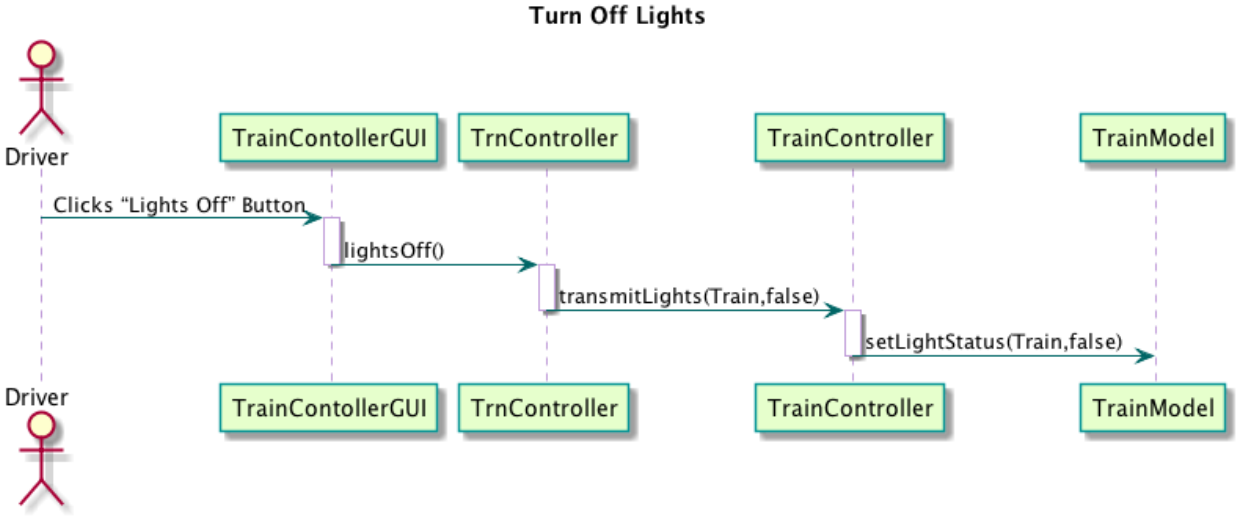
This sequence flow begins when the driver clicks on a button to deactivate a type of brakes on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController deactivate the proper brakes. Then the TrnController sends the new state of the brakes, the boolean value "false", to the TrainModel through the TrainController, using the train's identifier string to ensure that the proper model object is modified.

4.5.5.6 Turn On Lights



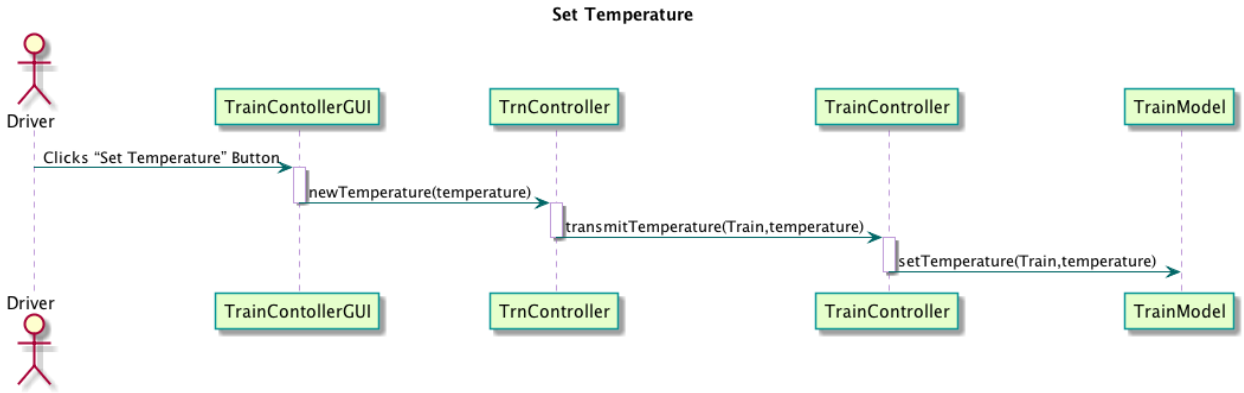
This sequence flow begins when the driver clicks on a button to turn on the train's lights on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController turn on the lights. Then the TrnController sends the new state of the lights, the boolean value "true", to the TrainModel through the TrainController, using the train's identifier string to ensure that the proper model object is modified.

4.5.5.7 Turn Off Lights



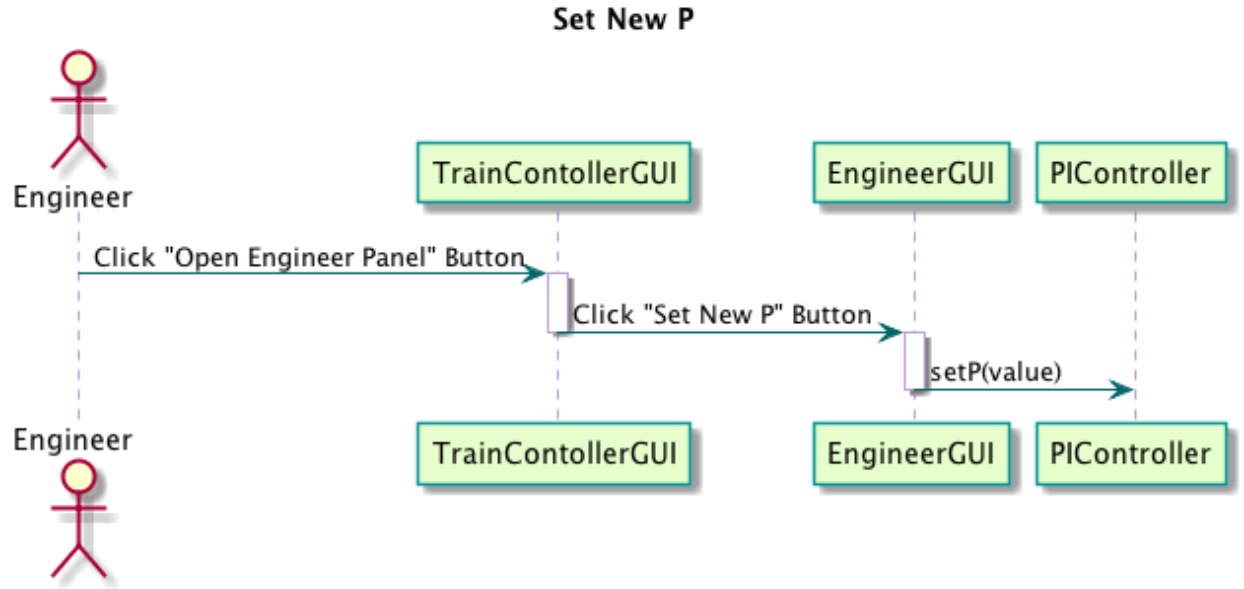
This sequence flow begins when the driver clicks on a button to turn off the train's lights on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController turn off the lights. Then the TrnController sends the new state of the lights, the boolean value "false", to the TrainModel through the TrainController, using the train's identifier string to ensure that the proper model object is modified.

4.5.5.8 Set Temperature



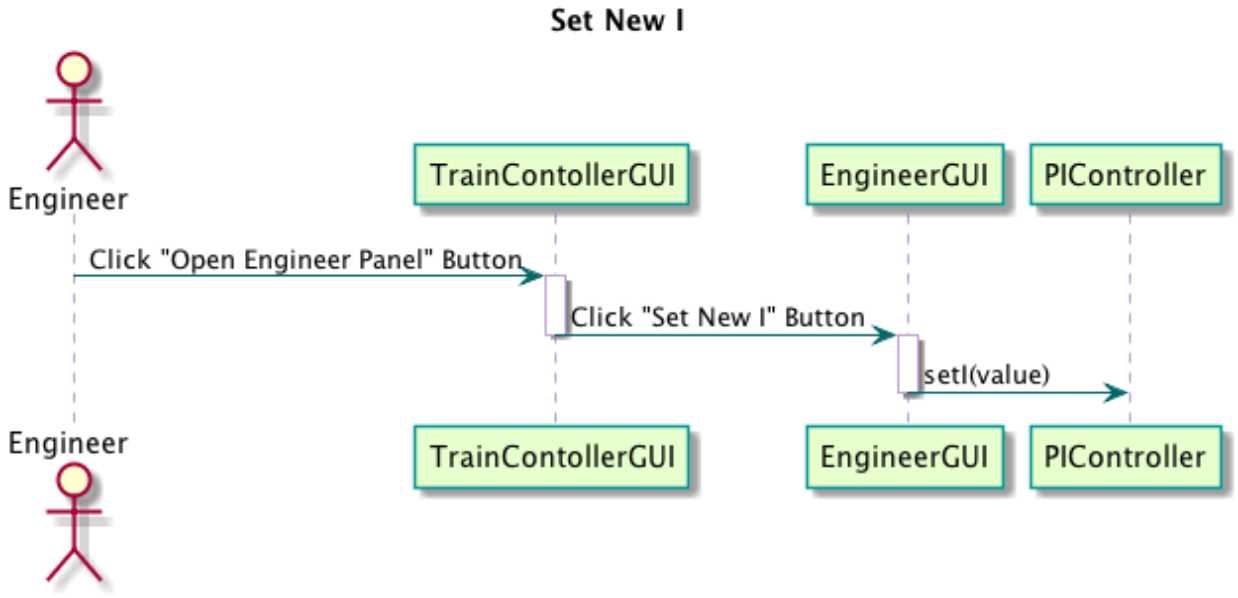
This sequence flow begins when the driver enters a new temperature value and clicks "Set" on the TrainControllerGUI. The TrainControllerGUI requests that the TrnController modifies its temperature value. Then the TrnController sends the new temperature value to the TrainModel through the TrainController, using the train's identifier string to ensure that the proper model object is modified.

4.5.5.9 Set New P Value



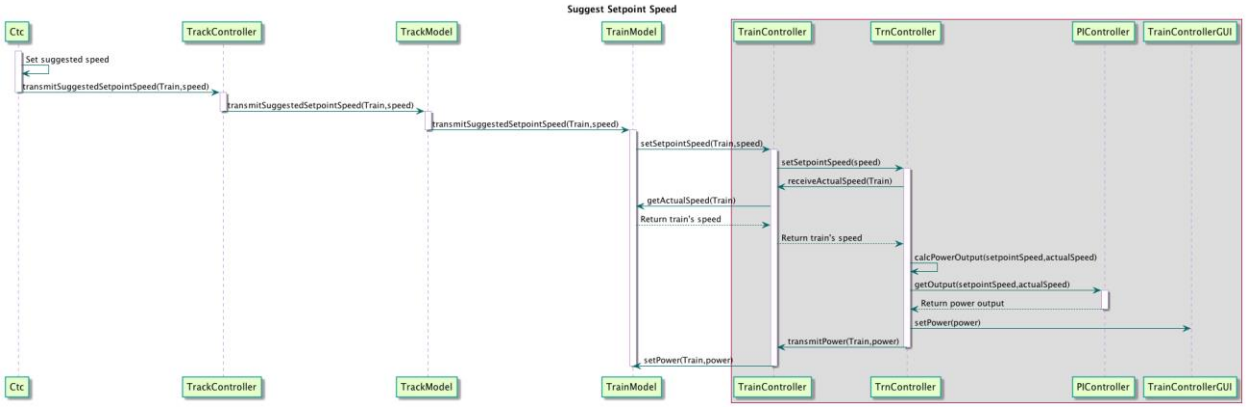
This sequence flow begins when the yard's train engineer opens the EngineerGUI from the TrainControllerGUI, enters a new P value, and clicks the "Set New P" button. The EngineerGUI will set the P value of the train's corresponding PIController object to the new value entered by the engineer.

4.5.5.10 Set New I Value



This sequence flow begins when the yard's train engineer opens the EngineerGUI from the TrainControllerGUI, enters a new I value, and clicks the "Set New I" button. The EngineerGUI will set the I value of the train's corresponding PIController object to the new value entered by the engineer.

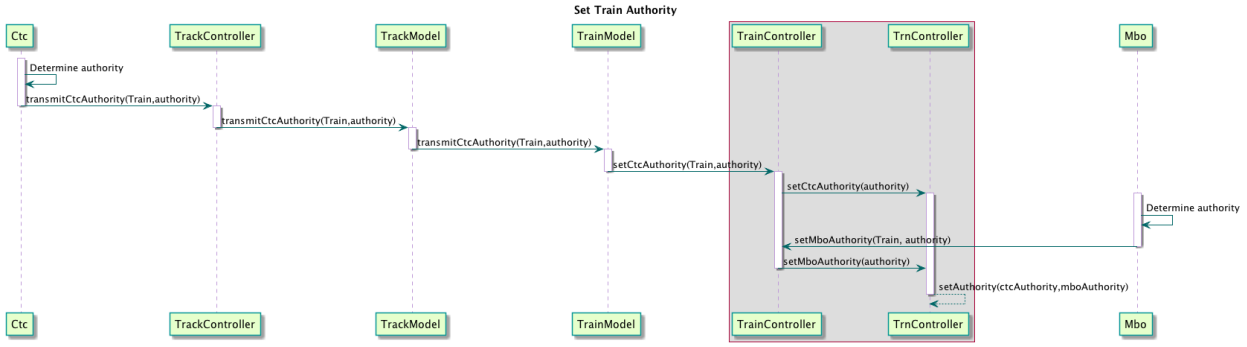
4.5.5.11 Suggest Setpoint Speed



This sequence flow begins when the CTC suggests a new setpoint speed for a particular train. This setpoint speed is transmitted through the Track Controller, Track Model, and Train Model modules and finally sent from the Train Model to the TrainController object with the string identifier of the train. The TrainController object sets the setpoint speed of the proper TrnController object to the value sent from the CTC. Then the TrnController object requests the actual speed of its train, which the TrainController object gets from the TrainModel object using the train's identifier string. With the setpoint speed and actual speed of the train, the TrnController calculates a new power output by getting the output from the PIController object. The TrnController then sets the power display of the TrainControllerGUI so the driver can see the new power value. Finally, the TrnController transmits the new power to the TrainModel through the

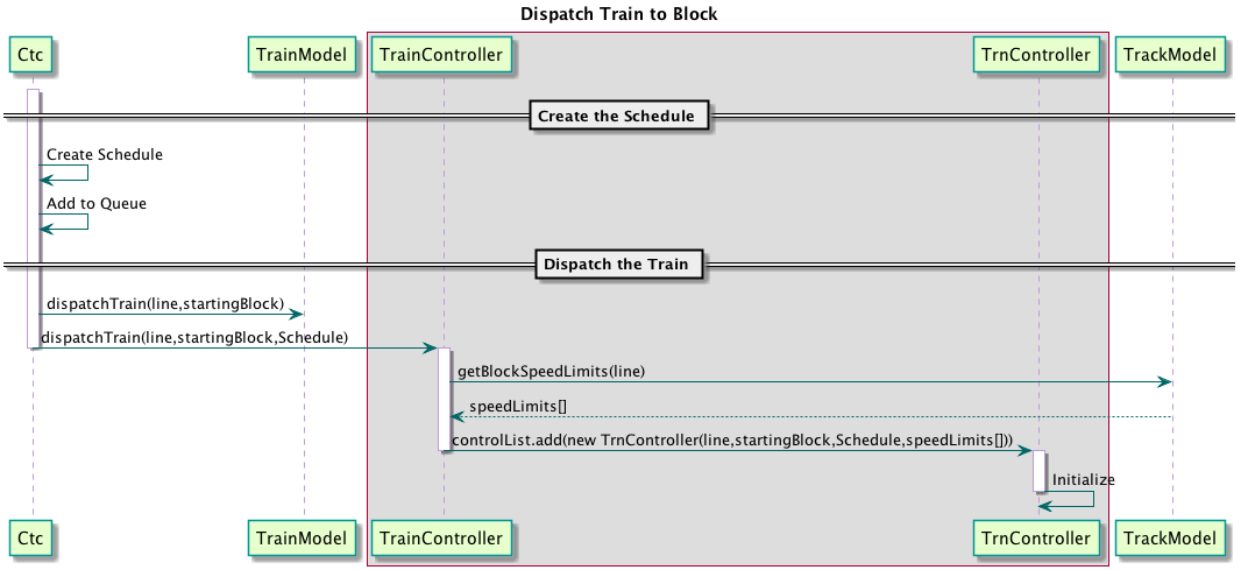
TrainController, using the identifier string of the train to make sure the proper model object is modified.

4.5.5.12 Set Train Authority



This sequence flow begins when either the CTC or the MBO calculates a new authority for a train, which occurs on every cycle of the system's clock. The authority from the CTC is transmitted through the Track Controller, Track Model, and Train Model modules, and then passed through the TrainController object to the proper TrnController object with the train's identifier string. The authority from the MBO is simply transmitted directly to the TrainController object, which sets the MBO authority value of the proper TrnController object with the train's identifier string. Finally, the TrnController object calculates the overall authority at which it must come to a stop by taking the lesser of the MBO and CTC authorities.

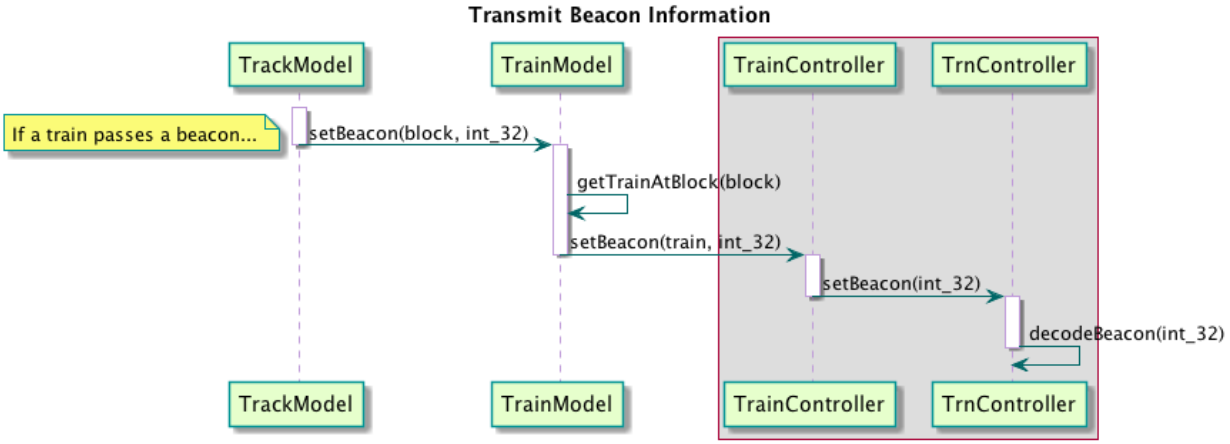
4.5.5.13 Dispatch Train to Block



This sequence flow begins when the dispatcher using the CTC module dispatches a new train. The CTC will inform the TrainController object that a new train has been dispatched by executing the *dispatchTrain()* function, passing relevant information as arguments. The TrainController will then fetch the list of speed limits of the line that the train will be operating on from the Track Model module. Finally, the TrainController object will add a new TrnController object to its list of controllers, and the new

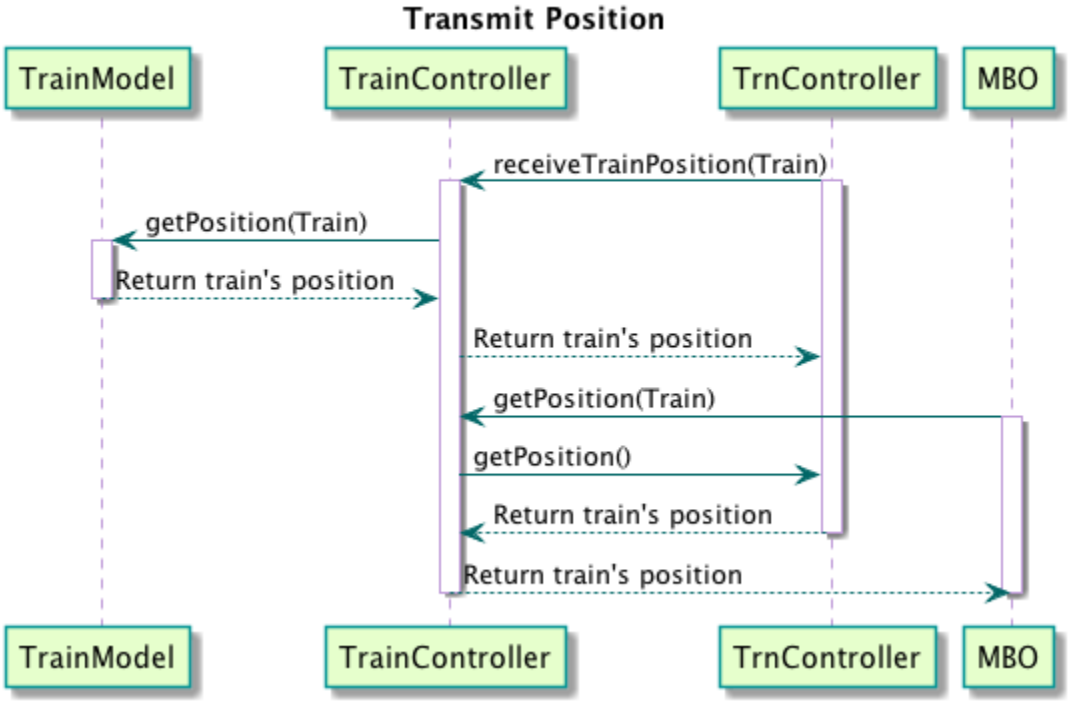
TrnController will initialize itself with all the relevant information so the train can begin operation.

4.5.5.14 Transmit Beacon Information



This sequence flow begins when a train passes a beacon on the track. If the Track Model detects a train on a block with a beacon, it will notify the Train Model of the block and the beacon value. The Train Model will then determine which train is on that block. It will then notify the TrainController object of the train's identifier string and the beacon value. The TrainController will find the TrnController with that identifier and set its beacon value. Then the TrnController can interpret the value to recover any information that was encoded in the beacon.

4.5.5.15 Transmit Position

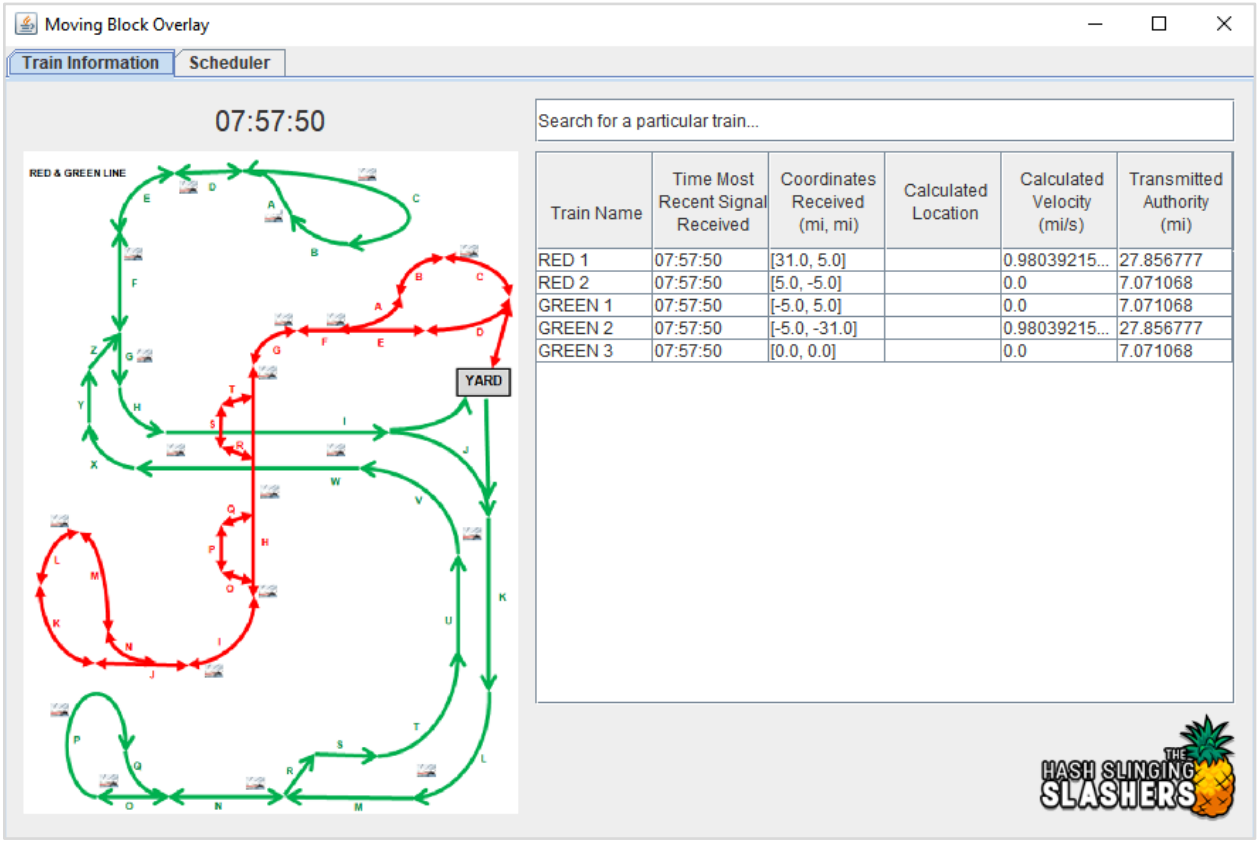


This sequence flow occurs on every cycle of the system's clock. The position of the train originates in the Train Model where the GPS is located. The TrnController object

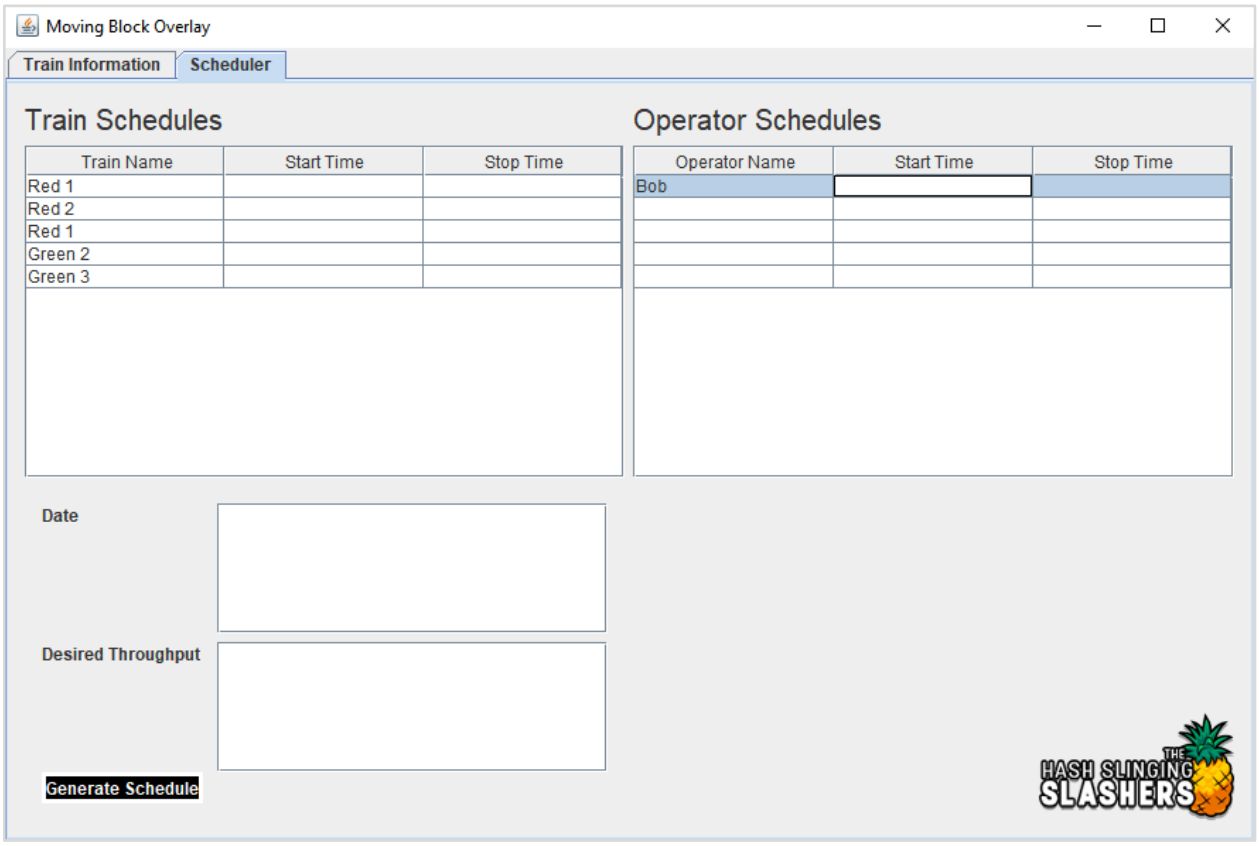
will request its position every clock cycle, which the TrainController will get from the Train Model module using the train's identifier string. Once the TrnController has its particular location, it can use it to determine the speed limit of the track section it is currently on. The MBO will also need the position of each train. The MBO will get the position of each TrnController by going through the TrainController object, passing each train's identifier string as an argument so the TrainController gets the position from the correct TrnController object in its list.

4.6 MBO

4.6.1 INTERFACE



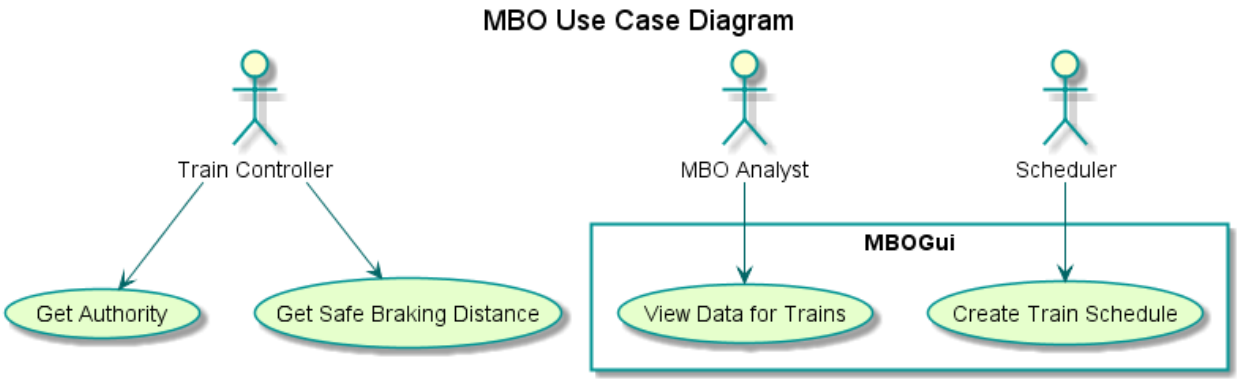
Above is the preliminary design of the graphical user interface (GUI) of the MBO. The interface design will change by the time the system is a finished product, but this preliminary interface shows all the important functionality of the GUI. The GUI has two view modes, "Train Information" and "Scheduler". In "Train Information" mode, the user can view the information for all trains in the system, and can search for a particular train by entering a regular expression in the search bar that matches the name of the train.



In “Scheduler” mode, the user can enter the start and stop times for all trains in the yard, as well as the names, start times, and stop times for a given set of train operators. The user can also enter a string describing when the schedule should be used (e.g. “Sep 6 2002” or “Sunday Schedule If Eugene Is Off”), and a double representing the desired throughput of the system. The user can then attempt to generate a schedule by clicking the “Generate Schedule” button. Note that schedule generation will fail if the start and stop times for the trains are incomplete, if any operator schedules are incomplete, or the desired throughput is impossible with the given train and operator schedules.

4.6.2 USE CASE

The following diagram shows the actions that the users of the MBO can perform through the module. The Train Controller interacts with the MBO directly, while the MBO analyst and scheduler users must interact with the module through the MBOGui. The driver can perform actions through the TrainControllerGUI and the engineer can perform actions through the EngineerGUI. GUI interactions are thus seperated from code interactions in the diagram. Each use case is explained in more detail below the diagram. The sequence flow for each of these use cases, as well as the flow for several internal functionalities not controller by either of these users, is detailed in Section 4.5.5 of this document.



4.6.3 GET AUTHORITY

The Train Controller will receive vital authority from the MBO after the MBO queries it for its current position. See Section 4.6.5.1 for corresponding sequence diagram.

4.6.4 GET SAFE BRAKING DISTANCE

The Train Controller will receive a safe braking distance from the MBO after the MBO queries it for its current position. See Section 4.6.5.2 for corresponding sequence diagram.

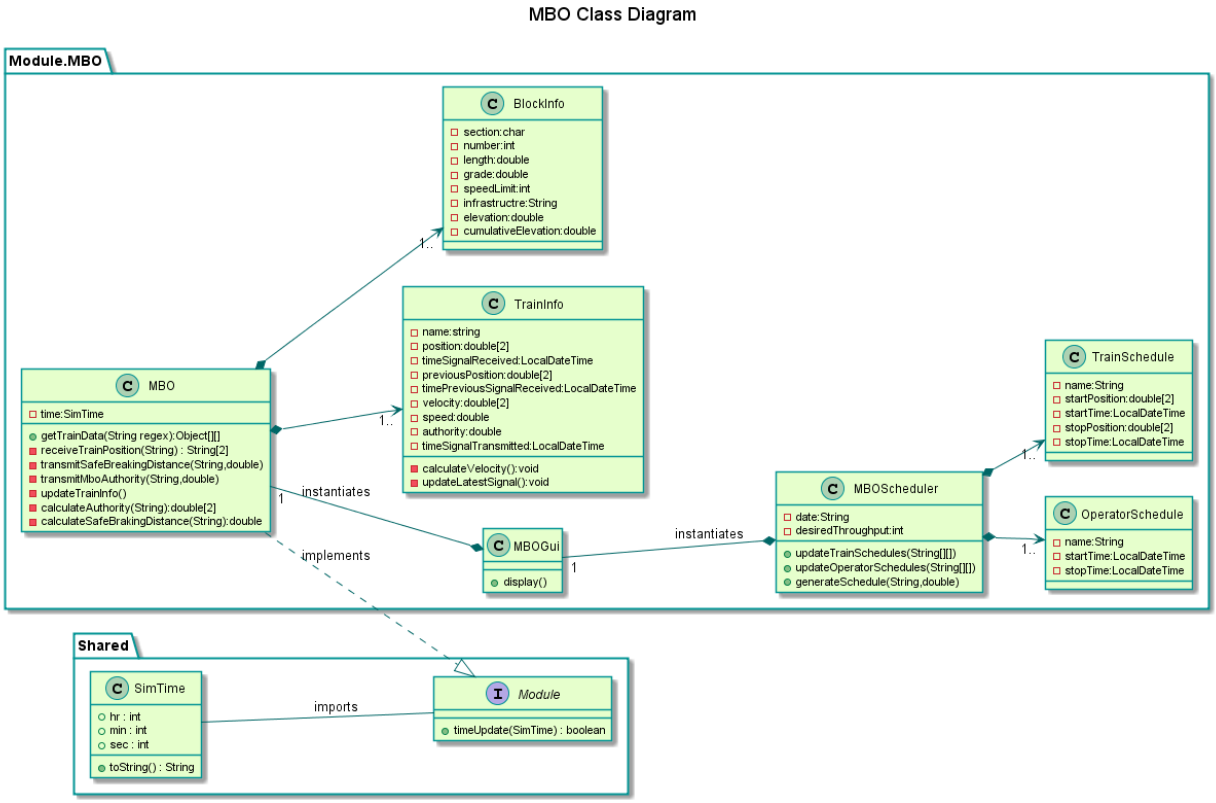
4.6.5 VIEW DATA FOR TRAINS

The MBO analyst can view the data for trains through the GUI, which is updated in real time. The user has the option to view all the trains at once or search for a particular subset. See Section 4.6.5.3 for corresponding sequence diagram.

4.6.6 CREATE TRAIN SCHEDULE

Through the GUI, the Scheduler has the ability to enter the start and stop times for all trains in the yard, as well as the names, start times, and stop times of the train operators. The user can also enter a short description of the schedule and the desired throughput. When the Scheduler clicks the “Generate Schedule” button, the MBOScheduler attempts to generate a schedule with the desired parameters and will save it to a file if successful and display an error otherwise. See Section 4.6.5.4 for corresponding sequence diagram.

4.6.7 CLASS DIAGRAM



The class diagram for the MBO is shown above. The main class within the Module.MBO package is MBO. This class implements Module, connects to other modules, instantiates the MBOGui, contains the aggregation of of TrainInfo and BlockInfo objects, and is responsible for receiving position signals and transmitting authority and safe braking distance. Each of these five main functionalities are described below:

- Implements Module - Module is implemented (just as in every other North Shore Extension module) to contractually require the timeUpdate(SimTime) method within the Track Model. As described in Section 3, this function is meant to serve as a “rising clock edge” signalling the passage of one virtual second to the module. This function will be used within the MBO module to repaint the GUI, timestamp received and transmitted signals, and calculate the velocity of trains.
- Connects to other modules - The MBO class has references to the Train Controller main class. This cross-reference will be set by the Simulator immediately after all modules are initialized, as shown in the Simulator state diagram in Section 3. The reasonings for providing direct communication routes to these modules are as such:
    - Simulator - This path will be used to start/pause, as well as control the simulation speed from within the MBO class.
    - Train Controller - This path will be used to receive position signals from and transmit vital authority and safe braking distance signals to the Train Controller.
- Instantiates the MBO GUI - The MBO will create a single MBOGui on initialization by the simulator, which will serve as a hub for the MBO user to interact with the

train system. The MBOGui will have a reference back to the MBO to communicate with the track status and block characteristics, and will instantiate its own MBOScheduler in order to generate train schedules for the yard

- Data aggregations - The MBO class relies on two private classes to store information about the trains in the yard and the layout of the track.
    - TrainInfo - The TrainInfo class represents a single train in the yard and includes the current position, previous position, calculated velocity, calculated authority, and calculate safe braking distance.
    - BlockInfo - The BlockInfo class represents a single block on the track and characteristics inherent to that block, including block length and speed limit.
- Receiving position signals and transmitting safe braking distance - The MBO queries the TrainController for the positions of the trains in the yard and subsequently updates its values for the authority and safe braking distance of these trains. The MBO takes the active role at all points in this process in order to ensure that all trains have sufficiently conservative authorities and braking distances at all times.

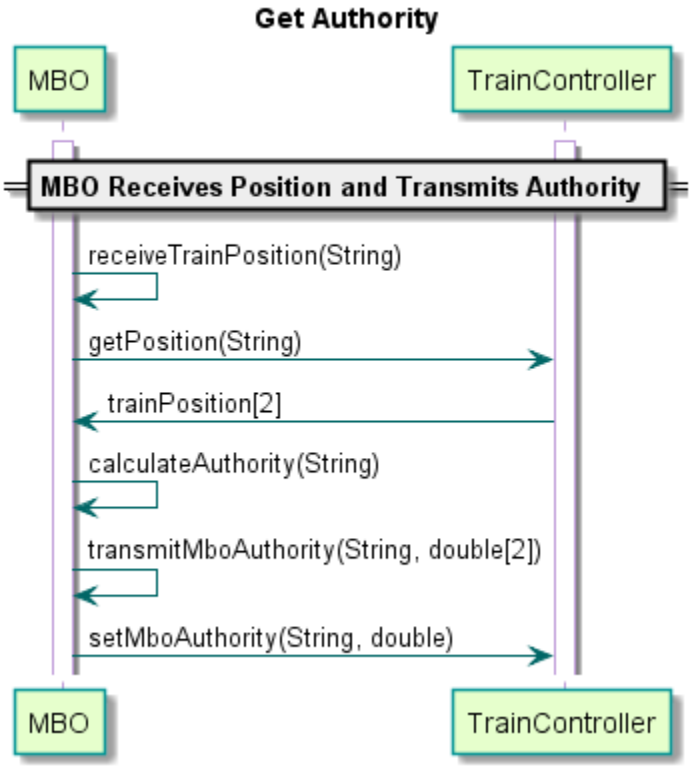
4.6.8 DATA DESIGN

Data Being Stored	Proposed Data Structure	Rationale
List of TrainInfo	ArrayList<TrainInfo>	An ArrayList will allow the MBO to quickly and easily iterate over all its trains when receiving signals, calculating values, and transmitting signals
Position of the train	double[2]	The most logical implementation for storing an (x,y) coordinate is in a double array of size 2.
Velocity of the train	double[2]	The velocity of the train can be represented as a 2-dimensional vector, which in turn is easily stored in the form of a double array of size 2
Individual train and operator schedules	Object[ ][ ]	The MBOGui receives the individual train and operator schedules from the user in the form of an Object[][], which would then allow the MBOScheduler to easily obtain names and times and use these to

		generate a schedule for the yard

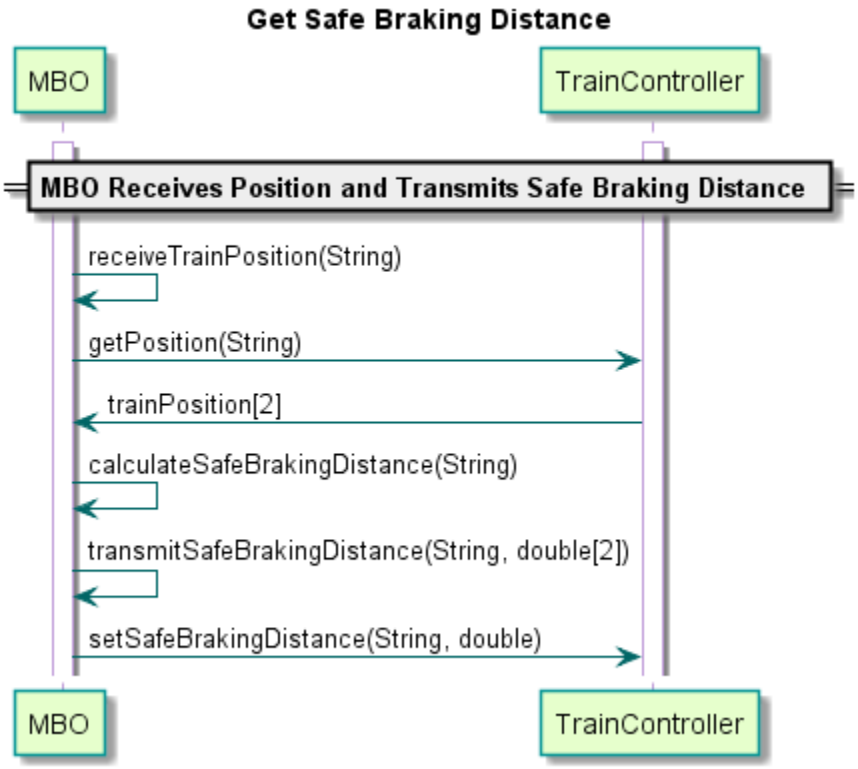
4.6.9 SEQUENCE DIAGRAMS

4.6.9.1 Get Authority

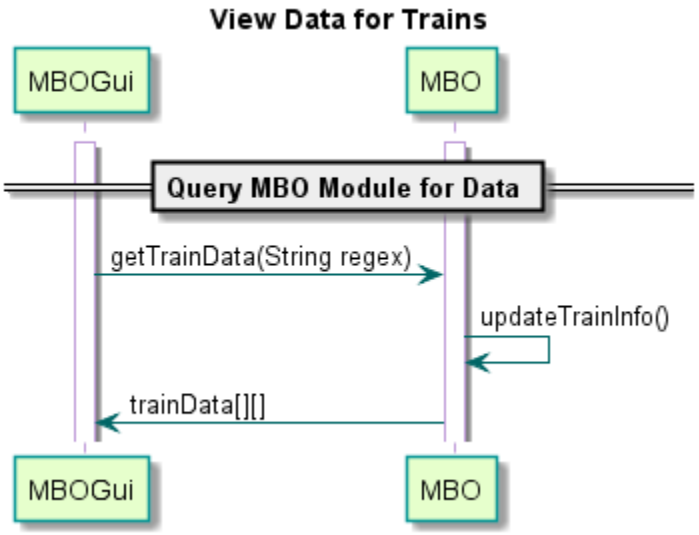


This sequence flow begins on every cycle of the system's clock. The MBOThe MBOGui queries the MBO for the most recent position, time position was received, calculated velocity, transmitted authority, and time authority was transmitted for all trains whose names match the **regex** argument. The MBO executes the updateTrainInfo() function to verify that all its data is accurate, and returns an Object[][] containing the train data to the MBOGui, which then displays it to the user.

4.6.9.2 Get Safe Braking Distance

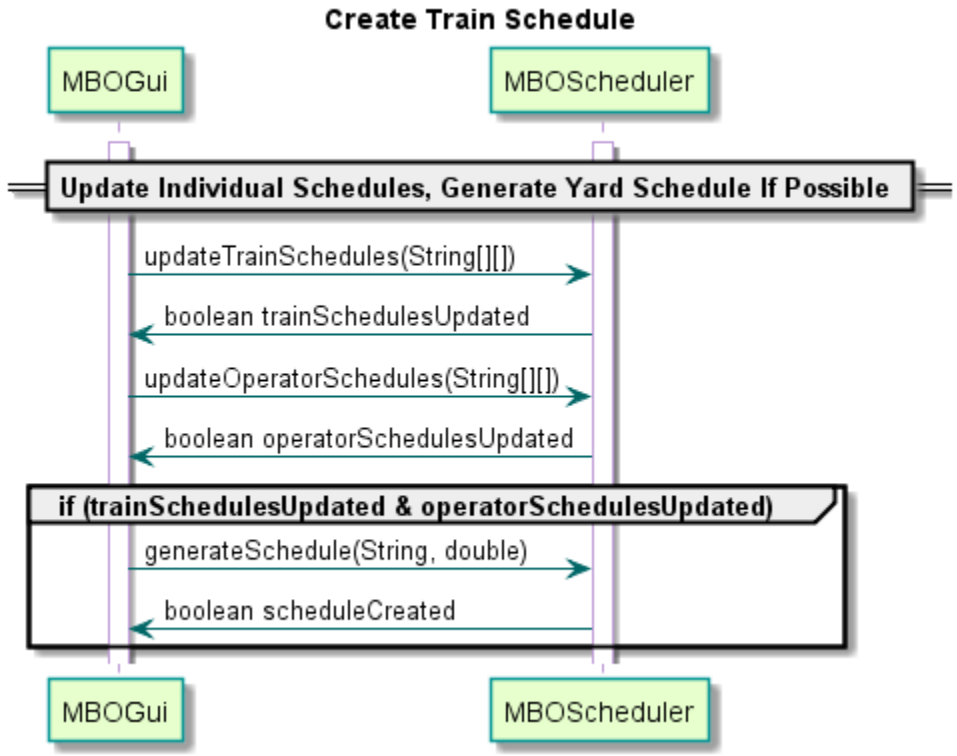


4.6.9.3 View Data for Trains



This sequence flow begins when the MBOGui updates its display, which occurs on every cycle of the system's clock. The MBOGui queries the MBO for the most recent position, time position was received, calculated velocity, transmitted authority, and time authority was transmitted for all trains whose names match the **regex** argument. The MBO executes the updateTrainInfo() function to verify that all its data is accurate, and returns an Object[][] containing the train data to the MBOGui, which then displays it to the user.

4.6.9.4 Create Train Schedule



This sequence flow begins when the user attempts to generate a new schedule by clicking the “Generate Schedule” button on the “Scheduler” tab of the MBOGui. The MBOGui first executes the updateTrainSchedules() function in the MBOScheduler module, passing the String[][] obtained from the user’s input as the argument. The MBOScheduler will then return the boolean trainSchedulesUpdated, which is True if the module was able to parse the information as needed and False otherwise. The MBOGui will then execute the updateOperatorSchedules() function, passing the user’s input as a String[][] to the MBOScheduler, and the MBOScheduler will return the boolean operatorSchedulesUpdated indicating whether the data is valid. If either of the return values are False, the MBOGui will generate an error message to the user. If both are valid, the MBOGui will execute the generateSchedule() function in the MBOScheduler module, passing as arguments a String representing the filename to save the schedule under and a double representing the desired throughput. The MBOScheduler will then attempt to generate a schedule for the yard with the desired throughput, and return a boolean scheduleCreated indicating whether it was successful.