# CS 351: Design of Large Programs
# Project 4: SmartRail Simulator

Brooke Chenoweth

Fall 2024

## Problem Description

SmartRail is a visionary system concept that allows trains to interact directly with the tracks, switches, and lights that a part of a light rail network. Each train is an independent agent, which selects a destination (normally based on a schedule, policy, or conductor input), discovers the route to the destination, and secures the route against possible collisions (by repositioning switches and setting lights). Trains are designed not to be able to run red lights. For a train to move, it needs to secure a route or a portion of it by locking lights and switches appropriately, both along the route itself and in a protection zone around the route. The latter is designed to prevent other trains from entering or crossing the route.

Trains function independently of each other and cannot communicate with other trains. A train is aware of the track on which it is located and, similarly, the track is aware of the identity of the train present at that location. Two trains cannot be located on the same track–a route consists of multiple tracks that connect with each other.

For modularity and flexibility reasons, each component of the light rail system is aware only of its immediate neighbors in the system.

## Simplifying Assumptions

- The stationary components are stations, switches, and track segments. The configuration is loaded on from a file using the format specified below.
- There are no rail crossings.
- Trains enter and exit the system at stations only.
- The user selects the destination for a train.
- A train can be heading either to the left or to the right. A train will continue heading in its current direction left/right until it reaches a station. (No reversing, no U-turns)
- Trains change direction only when at a station.
- All trains move at the same speed.

# Configuration File Format

- Text file with each line giving information about station, switch, or section of track. These lines can occur in any order, so you can interleave track and switch information, put station locations at the beginning or end of the file, etc.
- You may assume that location coordinates will range from 0 to 10. If you wish to support a broader range of locations (larger or negative values) document it in your readme so we know what to expect. (You may chose to support more, but just have your GUI get a bit ugly in that case.)
- Station line starts with "station" and then is followed by two numbers representing the x and y coordinates of the station. A station is located at the end of a single track segment.
- Switch line starts with "switch" and then is followed by two numbers representing the x and y coordinates of the switch. A switch connects two track segments on the left/right to a single track segment on the right/left.
- A track segment line starts with "track" and is followed by either four or five numbers. In both cases, the first four numbers represent the coordinates of the endpoints (x and y of first point, then x and y of second). Endpoints should be at locations of stations, switches, or other track specified somewhere in the file, though the other components may be specified after this track.
  - If there were only 4 numbers, this line represents a single track segment connecting the two endpoints.
  - If there were 5 numbers, the fifth number is the number of equally spaced segments of track connecting the two endpoints.
- You must verify the the correctness of a configuration file when loaded. (Components aren't dangling, track segments don't cross each other, etc.) You may chose to reject the file entirely or attempt to recover from a bad configuration. Document your choices in the readme file so we know what to expect.

## Program GUI

- Program initially reads configuration file to set up simulation. This file must be specified as a command line argument. Document this and any other command line arguments in your readme file.
- User selects initial starting station and destination station for at least one train. Make sure you specify how to do this in your readme, especially for more complicated scenarios involving multiple trains.
- Display should have track segments, stations, switches, and trains clearly displayed and updated as the simulation progresses.

- Stationary elements (stations, switches, track) should indicate if they are in a locked or free state.
- Train location should be updated as it moves and should indicate status of idle, seeking path, or moving (at least) somehow. (Additional states are fine, of course.) Document the meaning of any colors and/or symbols displayed so we can interpret your simulation.

  For consistency in grading, please use the following colors for your train status. (Exact shades are up to you.)

| **Train Status** | **Color** |
|---|---|
| Idle | Gray |
| Seeking path | Yellow or Orange |
| Locking path | Red or Purple |
| Moving | Green |

- Somehow indicate if train was unable to successfully find and secure the desired path.
- Simulation should be displayed at a human preceptible speed, not just have trains teleporting from station to station with no stops displayed in between. It is fine to have the train jump from one component to the next neighboring component on the path rather than smoothly animating it.

  That said, your simulation logic should not depend upon timing induced by making threads sleep. (So, okay to sleep to slow down enough for humans to see, but you should have more or less the same simulation result if you turn off the the sleeping.)

# Implementation Details

- Each component (train, tracks, stations, etc.) is an active object running on its own thread.
- Each component can only talk to its immediate neighbors. (You should not call a method on a neighboring object that calls a method on the next object that calls a method on the next object, and so on.) Pass some sort of message to the neighbor instead and let the neighbor handle the message on its own thread.
- Make sure you are using appropriate concurrent data structures and synchronization to avoid deadlock and race conditions. Suggestion: give every object a BlockingQueue of messages as an "inbox" and just have everything for that object coordinated through processing the messages one at a time.