I. Introduction

Throughout our lives, transportation has been an integral part of our daily activities. Progressive countries such as Japan, Hong Kong and Singapore have implemented their own subway systems – through automated rail transport, which is considered among the most efficient means of mass transportation, in order to address the growing needs of the commuting populace.

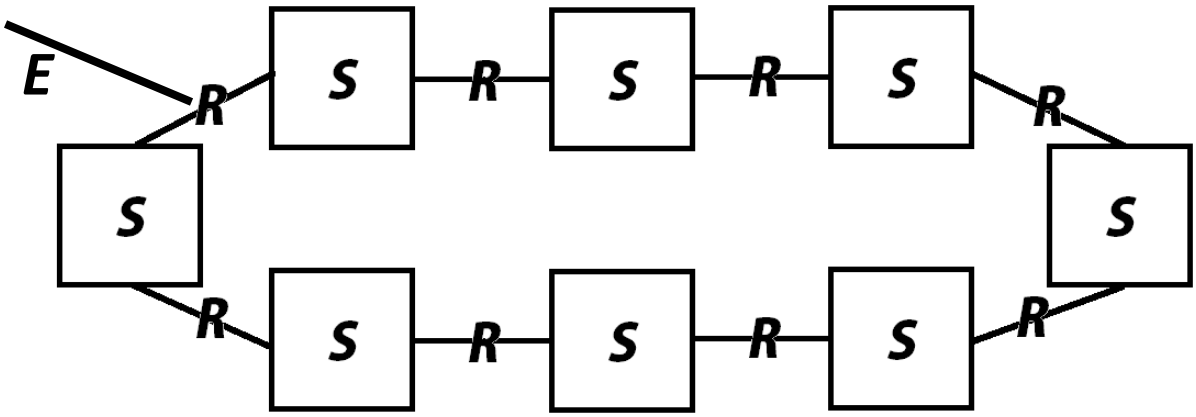
Unfortunately, the same can not be said for the Philippines. Overpopulation, unfinished infrastructure, lack of coordination and logistics all play a part in the country's stagnating and dismal state of transit. An article from The Philippine Star [1], dating October 20, 2016, stated that Manila ranked 10th on a list of cities with the worst traffic management in the world. This was in accordance with a website that collated global information on cost of living, crime rate and pollution. According to the same source, the Japan International Cooperation Agency (JICA) reported that traffic in the Philippines costs Php 2.4 billion daily towards the developing country's economy. Further, JICA warned that the Philippines is expected to lose Php 6 billion daily due to traffic by year 2030 should the state of transportation remain in its current state.

In order to travel across and within the bustling capital that is Metro Manila without relying on public utility jeepneys, taxi cabs and buses subject to the aforementioned heavy state of traffic, the next most efficient solution would be to board one of the the Light Rail Transit (LRT) lines or the Metro Rail Transit (MRT) should one's destination be in relative distance to the transits' multiple stations distributed along the metropolis. However, the rail transit services within the capital are plagued by customer complaints daily due to the inefficiency of the system used in their operations. An article by Romualdez [2], published on August 2014, revealed that the main cause for the problems, including slow loading of passengers, transit vehicles having to wait in the middle of railways and extreme crowding of stations, is the constant failure to follow coordination procedures.

CalTrain II, a course project on computer process synchronization, is an attempt at recreating the optimal work environment for effective rail transit within a community. In addition to being able to run a mock transit flow between eight (8) stations housing a total of fifteen (15) trains, the project is also able to simulate passenger count fluctuations, as well as their simultaneous boarding and alighting of trains. The end goal of the project is to show how busy and full train stations are to operate systematically in order to optimize their services. Additionally, the group believes that CalTrain II would be able to serve as a guide to its real life counterparts in the development and improvement of the public rail transit system of the country.

II. Project Design

The map layout in the project was implemented using a framework containing seventeen (17) segments. Eight (8) of them are stations, eight (8) are railroads connecting the said stations, and an extra one (1) segment which serves as the entry point for the trains. Internally, the map is implemented as a one-way circular linked list with an extra segment jutting into the segment before the first station. Trains, each of which has a length equal to the length of the segment it is currently on, are distributed among the segments. At maximum, fifteen (15) segments will each contain one train, leaving one (1) segment unoccupied by any train. This is done in order for the trains to be able to move around the map, as having all segments contain trains would result in a deadlock situation. The trains move in a counter clockwise direction. This is shown in the sample illustration below:



LEGEND:

S - Station

R - Railway

E - Entry

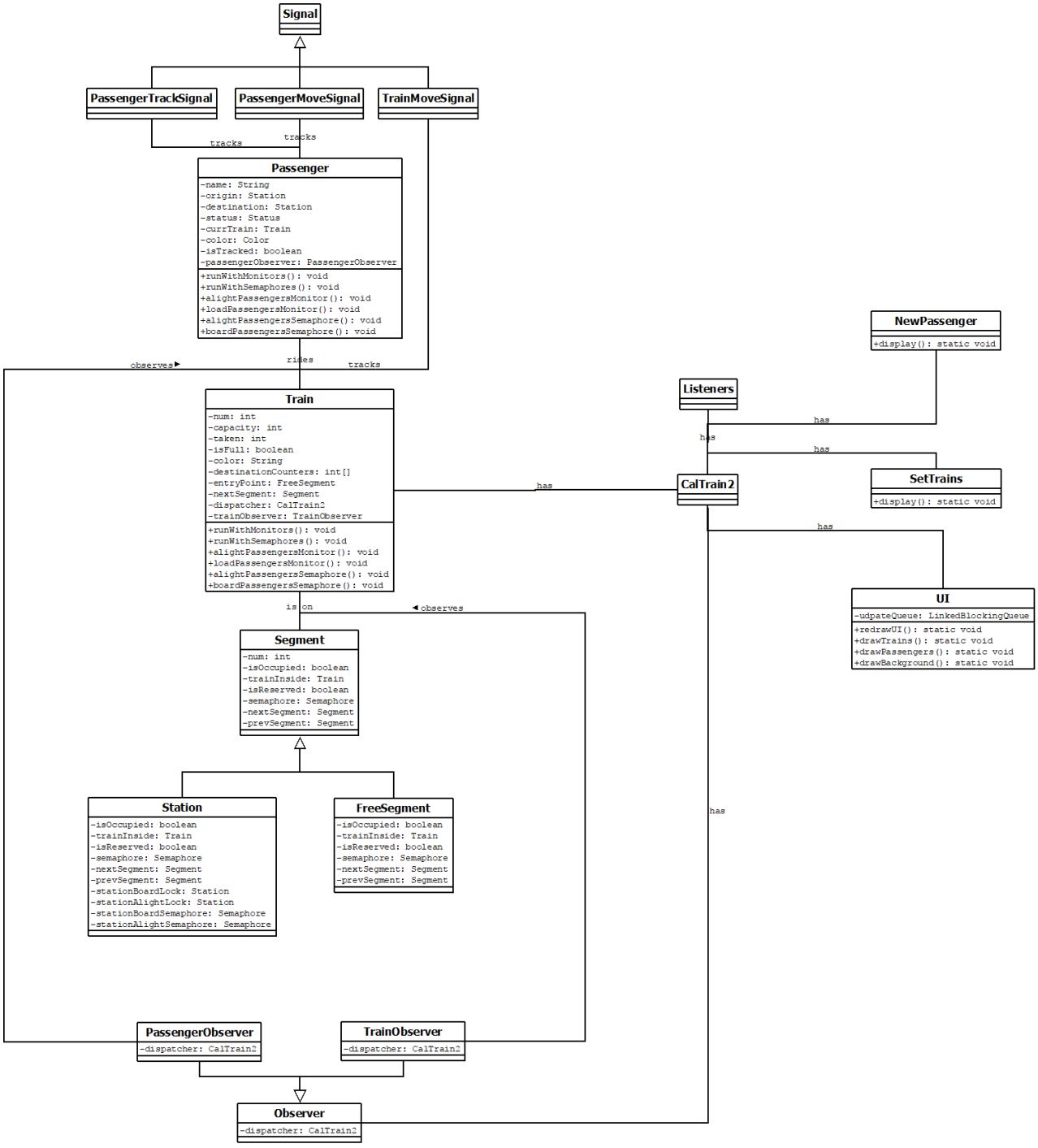
Design Patterns

The researchers have decided to use the **MVC framework** in order to differentiate between the user interface and the application logic for easier development. Moreover, the JavaFX framework *encourages* the use of the MVC framework.

A major design pattern used was the **observer pattern** due to the way the user interface is refreshed. The observer pattern was used in two ways, the **model observer framework** and the **interface observer framework**. The model observer pattern *listens* to changes made by the model classes. Once an event has registered, the appropriate action is then dispatched into the Controller class then finally into the interface observers who then interpret the signal and then perform the appropriate response to the user interface. This use also stems from the fact that the JavaFX framework **restricts the modification of the user interface to JavaFX threads *only***. That is, one cannot modify the user interface in threads other than the JavaFX application threads.

Finally, **facades** were used to abstract the JavaFX stage classes from the developers. The stage classes were wrapped by a class decorator which is treated as if it were a normal class with a static method.

Class Diagram


Signal
PassengerTrackSignal
PassengerMoveSignal
TrainMoveSignal
tracks
tracks
Passenger
-name: String
-origin: Station
-destination: Station
-status: Status
-currTrain: Train
-color: Color
-isTracked: boolean
-passengerObserver: PassengerObserver
+runWithMonitors(): void
+runWithSemaphores(): void
+alightPassengersMonitor(): void
+loadPassengersMonitor(): void
+alightPassengersSemaphore(): void
+boardPassengersSemaphore(): void
observes
rides
tracks
Train
-num: int
-capacity: int
-taken: int
-isFull: boolean
-color: String
-destinationCounters: int[]
-entryPoint: FreeSegment
-nextSegment: Segment
-dispatcher: CalTrain2
-trainObserver: TrainObserver
+runWithMonitors(): void
+runWithSemaphores(): void
+alightPassengersMonitor(): void
+loadPassengersMonitor(): void
+alightPassengersSemaphore(): void
+boardPassengersSemaphore(): void
has
is on
observes
Segment
-num: int
-isOccupied: boolean
-trainInside: Train
-isReserved: boolean
-semaphore: Semaphore
-nextSegment: Segment
-prevSegment: Segment
Station
-isOccupied: boolean
-trainInside: Train
-isReserved: boolean
-semaphore: Semaphore
-nextSegment: Segment
-prevSegment: Segment
-stationBoardLock: Station
-stationAlightLock: Station
-stationBoardSemaphore: Semaphore
-stationAlightSemaphore: Semaphore
FreeSegment
-isOccupied: boolean
-trainInside: Train
-isReserved: boolean
-semaphore: Semaphore
-nextSegment: Segment
-prevSegment: Segment
PassengerObserver
-dispatcher: CalTrain2
TrainObserver
-dispatcher: CalTrain2
Observer
-dispatcher: CalTrain2
NewPassenger
+display(): static void
Listeners
has
has
has
CalTrain2
SetTrains
+display(): static void
has
UI
-udpateQueue: LinkedBlockingQueue
+redrawUI(): static void
+drawTrains(): static void
+drawPassengers(): static void
+drawBackground(): static void
has

Classes

Note: For further reference and the complete source code, please refer to the indices at the end of this document.

Train.java

The Train class represents a single Train object in the system. It is of significance that the actual mechanisms to traverse the circular linked list framework are delegated to a thread where all synchronization methods and implements are placed. At most, there should be fifteen (15) train objects in the system.

Internally, the trains are preliminarily stored as an array of fifteen (15) Trains (called the **fleet**), which are then later wrapped by an array of fifteen (15) Threads. These threads are then started **chronologically** from the first index ([0]) to the last index ([15]), denoting maximum capacity.

Passenger.java

The Passenger class represents a single Passenger object in the system. Like the Train class, the actual mechanism to move itself around is delegated to a thread. There are no limits on the number of passengers created. However, once a passenger arrives in its destination, the thread is killed by the Controller.

Segment.java

The Segment class is a generic abstract class denoting a single segment in the system. There are two types (subclasses) of Segments:

1) **FreeSegment.java**, denoting a free segment (a segment in the system where the train should not board and alight passengers), and
2) **Station.java**, denoting a station segment where passengers can embark and disembark the train.

Observer.java

The Observer class is a generic abstract class representing the observers for the main models Train and Passenger. Since the segments are just treated as frameworks who have no synchronization implement in themselves, obsevers for the segments are not included. The two observers are:

1) **TrainObserver.java**, and
2) **PassengerObserver.java**.

Signal.java

The Signal class is a generic, abstract, and placeholder class representing the three different kinds of signals emmited by the controller guided by the listening of the observers. These three signals are:

1) **PassengerMoveSignal.java**, representing a signal for when the passenger moves from one segment to another,
2) **TrainMoveSignal.java**, representing a signal for when the train moves from one segment to another (whether station or a free segment), and
3) **PassengerTrackSignal.java**, representing a special signal for when the user decides to track a single passenger. Such special requests need to be granted by a separate and more dedicated signal tracker other than PassengerMoveSignal.

CalTrain2.java

This class represents the entire system (akin to a control center) who manages, controls, and directs the operations of the CalTrain2 services. Being a controller, it acts as the interface between the objects, its actions and its synchronizations, and the visualization of such actions.

Listeners.java

This class only acts a servant for the CalTrain2 class. This class specializes in handling events both coming from the Model and View sides (whereas CalTrain2 takes a much more generalized approach).

The rest of the classes are solely for the purposes of the visualizations and are more closely tied to the JavaFX framework than the synchronization implements.

II. Process Synchronization

The developers have used three types of synchronizations on the project. These are the **train synchronization**, the **passenger synchronization**, and the **visualization synchronization**.

Monitors and Semaphores in Java

The Java programming language provides its own implementations of monitors and semaphores as synchronization constructs.

Monitors

Monitors are implemented by using synchronization blocks and the methods wait() and notify(). The statements in the synchronization block correspond to the critical sections of the program. That is, the statements in the synchronization block cannot interleave with other threads which are also synchronized on the same object specified in the synchronization block. wait() blocks the thread owning the critical section until another thread synchronized on the same object calls notify() on the said object. notify() is a method which signals blocked threads synchronized on the same object in the context of notify(). **If notify() is called with no wait()s waiting, the notification is not queued up and is simply discarded.**

Semaphores

Semaphores are provided in the java.util.Concurrent library. Semaphores are initialized with a specified number of permits. When a thread calls the acquire() method of the semaphore, the number of permits is reduced or the thread blocks **until** a permit is available. When a thread calls the release() method, a permit is made available to other threads waiting on the semaphore. **release() can be called in such a way that the semaphore exceeds the original number of permits**.

Synchronization implements

Train Synchronization

The train synchronization systems deal with synchronization **between the trains**. These synchronization systems are put in place to prevent trains from 1) occupying a single segment at the same time and 2) to allow an incoming train to enter the loop, even if it means cutting in line.

- **Using monitors**
- **Loop monitor**

If a train is in a station, then it must load and alight passengers. Each load and alight methods are given one (1) second each to make the visualization slower and easier to follow. In the solution, monitors in the train are assigned to probe the next segment as to whether it is reserved by another train. If it is reserved, then it must wait until it is not. If it isn't, then it should **immediately reserve** the next segment to avoid race conditions. Then the train goes to the next segment.

Entry monitor

Upon first entry, though, the train must first check if the entry segment is occupied by another train. If it is, then it must wait for it to be clear. Afterwards, it must then check if the first segment after the entry point is reserved. If it is, then the train must wait until it is not. If it isn't then it should **immediately reserve** the next segment to avoid race conditions. The train then enters the loop.

Using semaphores

Loop monitor

A semaphore guarantees mutual exclusion for the entire loop monitor. If a train is in a station, then it must load and alight passengers. Each load and alight methods are given one (1) second each to make the visualization slower and easier to follow. In the solution, semaphores in the train are assigned to probe the next segment as to whether it is reserved by another train. If it is reserved, then it must wait until it is not. If it isn't, then it should **immediately reserve** the next segment to avoid race conditions. Then the train goes to the next segment.

Entry monitor

Upon first entry, though, the train must first check by semaphore if the entry segment is occupied by another train. If it is, then it must wait for it to be clear. Afterwards, it must then check if the first segment after the entry point is reserved. If it is, then the train must wait until it is not. If it isn't then it should **immediately reserve** the next segment to avoid race conditions. The train then enters the loop.

An advantage of using semaphores is that loops need not be manually written by the developers (which could be prone to deadlocks). Moreover, synchronized blocks, wait()s, and notify()s are all abstracted away from the user.

Service Saturation Heuristic (SSH)

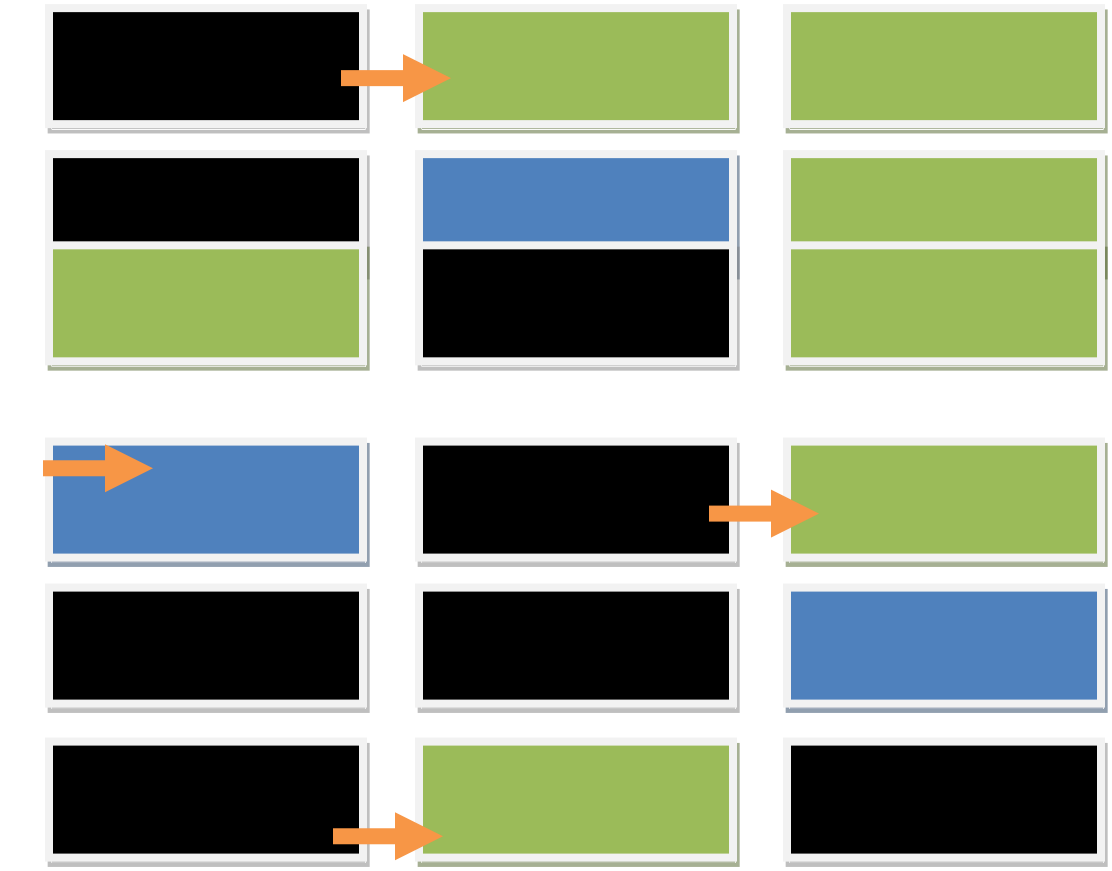
The first thing researchers should ask themselves is if a formula applies. According to the project specifications, the number of trains dispatched depends on the passenger demand. The developers have decided to use a heuristic to determine when to dispatch a train. The heuristic formula is as follows:

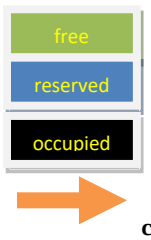
$$H_{ss} = \frac{|P|}{\sum_{T \in S} C}$$

Where $|P|$ is the total number of passengers in the system and $\sum_{T \in S} C$ is the total capacity of the currently dispatched trains.

The formula measures the **service saturation** of the system. That is, the ratio of the passengers and the total trains dispatched. $0 \leq H_{ss} < 0.75$ is the safe saturation level as decided by the researchers. Once the system reaches $H_{ss} \geq 0.75$, a new train will have to be dispatched. In situations with trains with small capacities or an unusually large surge of passengers, H_{ss} may even exceed 1 and signifies a **supersaturated system**.

Entry loop operation





check next segment if available

1) **Passenger synchronization**

Passenger synchronization deals with synchronizations between **trains, passengers,** and **stations**. The system is put in place to facilitate immediate boarding and alighting. A passenger has **five (5)** different states,

1) **Waiting**, where a passenger is waiting for a train **(note: queues are not honored)**,
2) **Boarding**, where a passenger is neither waiting for nor seated in a train,
3) **Seated**, where a passenger has settled in the train and is waiting for the destination,
4) **Alighting**, where a passenger is neither seated in nor out of the train, and
5) **Arrived**, where a passenger has now reached his/her destination.

Using monitors (passengers)

Waiting

A passenger waits on a monitor for boarding which is then signaled by the train. Once this monitor has signaled, the passenger tries to reserve a train seat. If he/she fails, he/she then waits again.

Boarding

A passenger finds his/her way to his/her seat.

Seated

The passenger settles and waits on his/her destination through a monitor for alighting. Once this monitor is signaled, the passenger gets up from his/her seat.

Alighting

A passenger finds his/her way out.

Arrived

A passenger has now exited by the train and has arrived in his/her destination.

Using monitors (trains)

A train first signals a station's alight monitor to notify the passengers waiting on that station that they've arrived. A one second delay was given to slow the graphics down and make the visualization easier to follow.

After the delay, a train then signals a station's board monitor to notify the passengers waiting to board on that station to ride the train. Another one second delay is then given for the same reasons as earlier.

Using semaphores (passengers)

Waiting

A passenger waits on a semaphore for boarding of which permit is then released by the train. Once this monitor has signaled, the passenger tries to reserve a train seat. If he/she fails, he/she then waits again.

Boarding

A passenger finds his/her way to his/her seat.

Seated

The passenger settles and waits on his/her destination through a semaphore for alighting. Once this semaphore is signaled, the passenger gets up from his/her seat.

Alighting

A passenger finds his/her way out.

Arrived

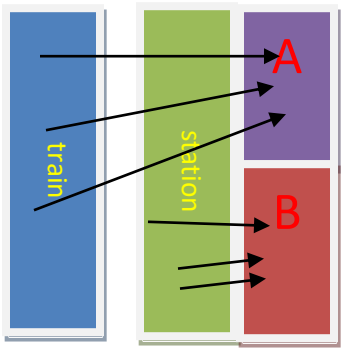
A passenger has now exited by the train and has arrived in his/her destination.

Using semaphores (trains)

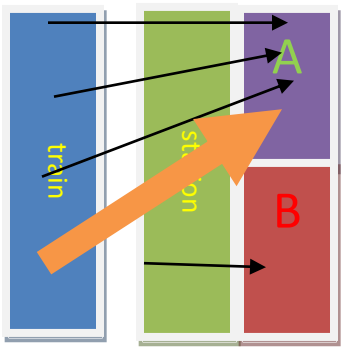
A train first releases permits to a station's alight semaphore to notify the passengers waiting on that station that they've arrived. A one second delay was given to slow the graphics down and make the visualization easier to follow.

After the delay, a train then signals a station's board semaphore to notify the passengers waiting to board on that station to ride the train. Another one second delay is then given for the same reasons as earlier.

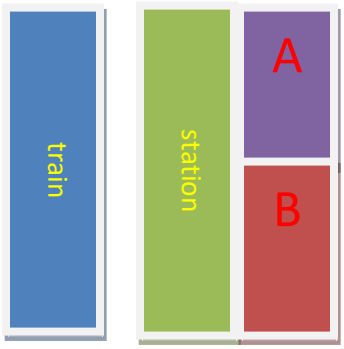
Passenger movement operation



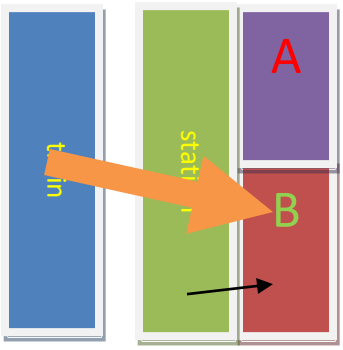
Train is in station, passengers to alight waiting on station



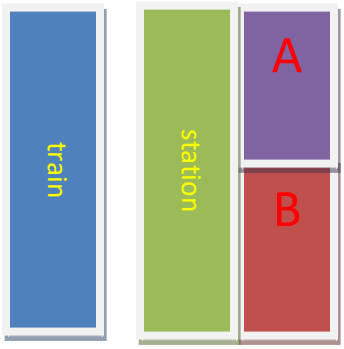
Train notifies the station's alight locks



Passengers now disembark



Train notifies station's board locks



Waiting passengers board the train, unless it becomes full.

2) **Visualization synchronization**

The need to synchronize the models with the interface requires **synchronizing the synchronizations to the interface** as well. Due to the scale of the requirements, the developers have decided to solely use monitors for the visualizations; these do not have any direct connection with the requirements anyway and is related to presentation rather than synchronization

III. Results and Analysis

A Priori Analysis

Based on the lectures given to us by the professor, monitors were more likely to be efficient compared to semaphores as a solution to the scenario presented. This is because the **wait()** function of a semaphore may be (trivially) implemented as follows:

```
void wait() {
  while (counter <= 0);
  counter--;
}
```

The line **while (counter <= 0);** results in a busy-wait. That is, an entire quantum of CPU time will be dedicated just for checking if a certain condition is met or not. This is wasteful as compared to just blocking the thread and then reawakening it again when it is ready in the case of monitors.

Through the lens of logical representation as follows:

|1|1 is stuck|2|1|2|1|3|1|3|1|4|1|4|1|5|1|5|1 is free to go|1|...| (Semaphore)

|1|1 is stuck|2|2|3|3|4|4|5|5|1 is free to go|1|1|...| (Monitor)

In the given illustration, the CPU process load distribution is replicated for both semaphores and monitors, each with five (5) competing threads. In both cases, thread 1 is to halt at time frame 2. Due to monitors managing all competing threads at the same time, it is able to relegate process 1 behind the other necessary processes which lead to more efficiency in their execution. On the other hand, the implementation with semaphores had to spend valuable resources in order check for whether or not thread 1 is ready to run in between the time frames which would be relegated to other threads and processes. This resulted in more time being used up in the long run, and in large scale processes, such as CalTrain II which has 15 trains and exponentially more passengers, this may lead to the bloating of temporary memory and the increase in the time needed in order to accomplish the same task at hand.

In addition to the mentioned delays, semaphore implementation has situations leading to deadlocks due to multiple semaphore – driven processes checking for run-time eligibility at a given time via the globally defined wait() method. On the other hand, monitor implementation has the thread switching facilitated by the monitor construct itself, and with the introduction of conditional variables, leads to less deadlocks and long pauses as

compared to their pure semaphore-driven counterparts.

Sample Runs and Benchmarks

Notes:

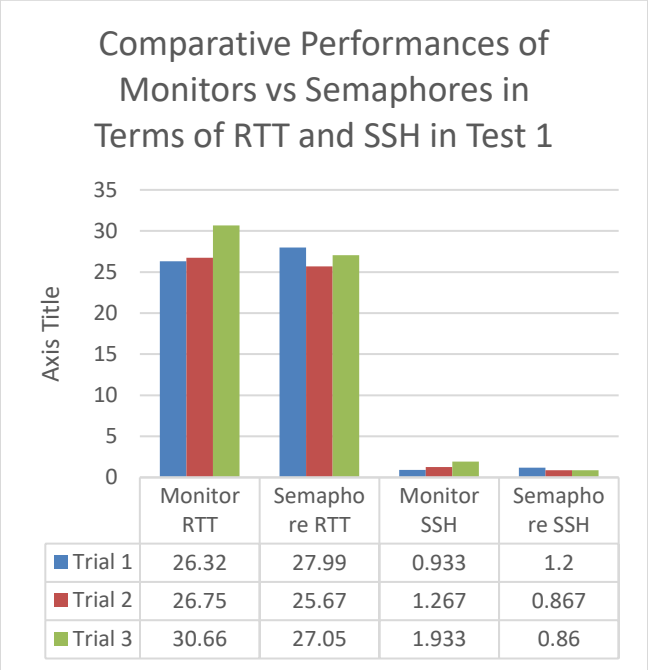
- Randomly generated passengers are just for benchmarking purposes. Explicit passenger tracking will be shown in the oral presentation. No other programs are opened aside from Microsoft Word and NetBeans to simulate real multi-thread scheduling behavior. The laptop on which the project was tested on was plugged in during testing.
- Trains only have one capacity to simulate overcrowded situations; the point is to **stress test** the synchronization mechanisms.

Definitions:

- **Roundtrip time (RTT)**, the time by which a passenger traverses the entire system exclusive of the starting station.

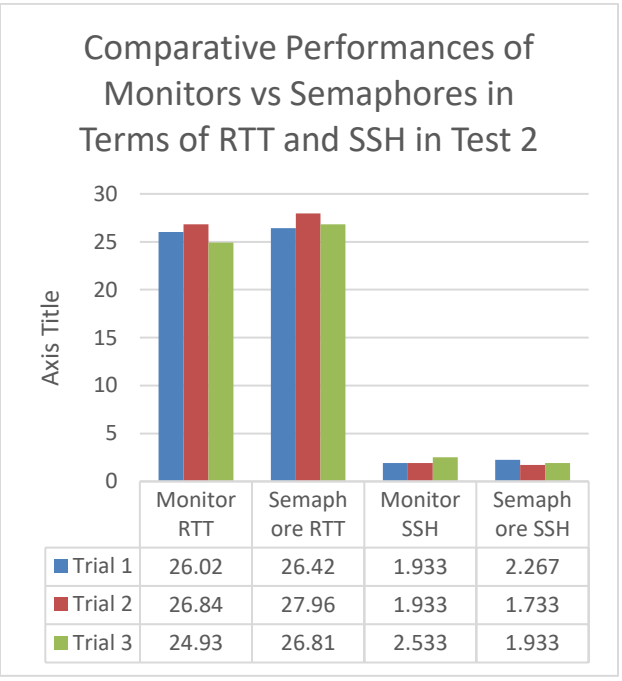
Test 1: Randomly distributed passenger load with a one-passenger train capacity (randomly generated at most every three seconds) – from station 1 to 8 after 30 seconds.

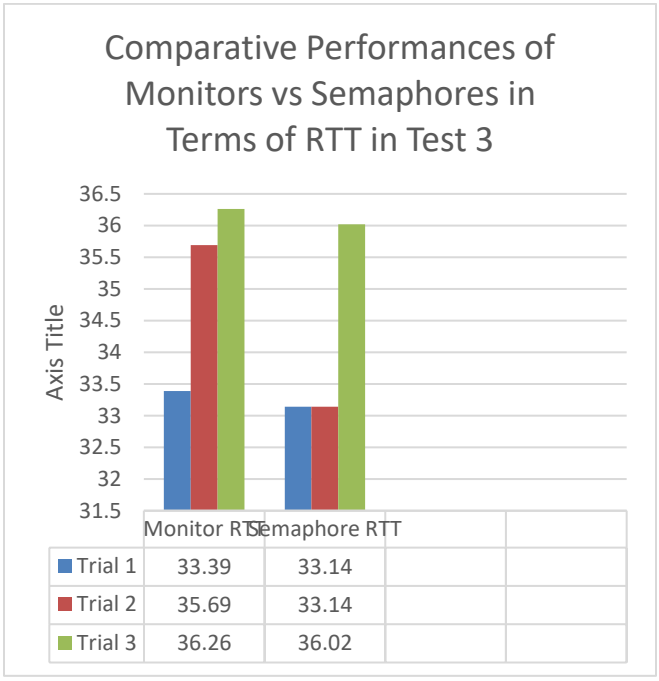
This test is designed to test how efficient the synchronization systems are with respect to a person traversing the system in the presence of other crowds.



1) **Monitors**
   a. **Roundtrip time:** 26.32 seconds **SSH after passenger arrives:** 0.933
   b. **Roundtrip time:** 26.75 seconds **SSH after passenger arrives:** 1.267
   c. **Roundtrip time:** 30.66 seconds **SSH after passenger arrives:** 1.933
   d. **Average RTT: 27.91 seconds**
      Average SSH: 1.377 seconds

2) **Semaphores**
   a. **Roundtrip time:** 27.99 seconds **SSH after passenger arrives:** 1.200
   b. **Roundtrip time:** 25.67 seconds **SSH after passenger arrives:** 0.867
   c. **Roundtrip time:** 27.05 seconds **SSH after passenger arrives:** 0.860
   d. **Average RTT: 26.90 seconds**
      Average SSH: 0.975 seconds





Test 2: Randomly distributed passenger load with a one-passenger train capacity (randomly generated at most every two seconds) – from station 1 to 8 after 30 seconds.

This test is designed to test how efficient the synchronization systems are with respect to a person traversing the system in the presence of larger crowds.

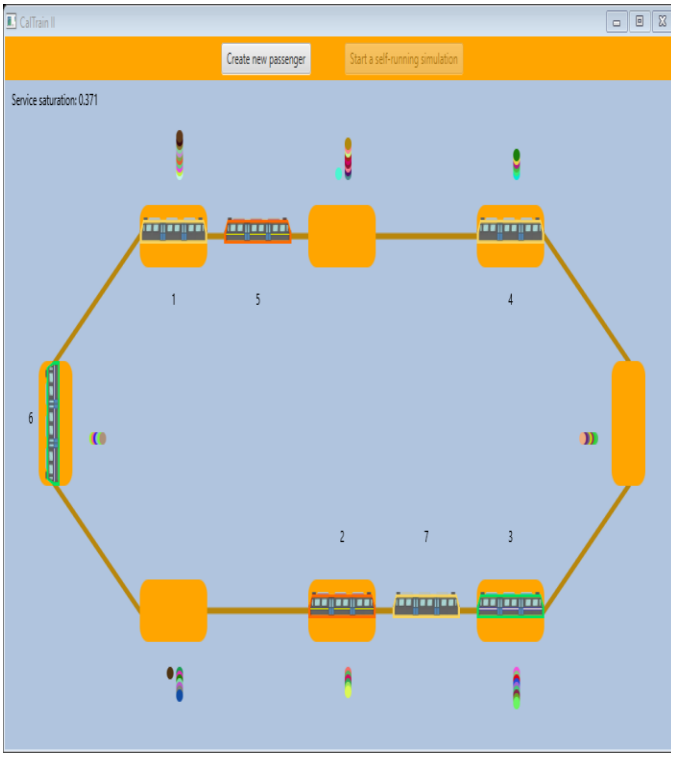
1) **Monitors**
   a. **Roundtrip time:** 26.02 seconds **SSH after passenger arrives:** 1.933
   b. **Roundtrip time:** 26.84 seconds **SSH after passenger arrives:** 1.933
   c. **Roundtrip time:** 24.93 seconds **SSH after passenger arrives:** 2.533
   d. **Average RTT: 25.93 seconds**
   Average SSH: 1.748
2) **Semaphores**
   a. **Roundtrip time:** 26.42 seconds **SSH after passenger arrives:** 2.267
   b. **Roundtrip time:** 27.96 seconds **SSH after passenger arrives:** 1.733
   c. **Roundtrip time:** 26.81 seconds **SSH after passenger arrives:** 1.933
   d. **Average RTT: 27.06 seconds**
   Average SSH: 1.977

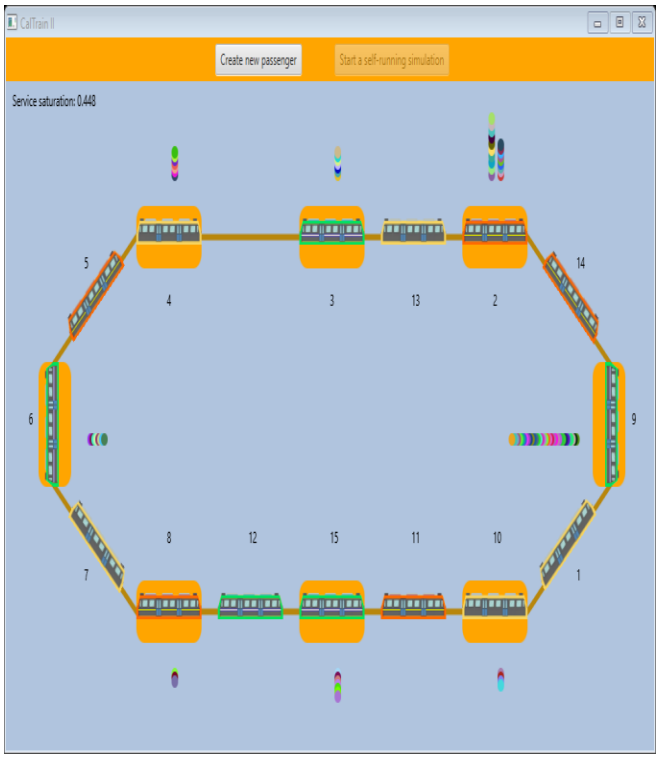
Test 3: Five-passenger surge test – from station 4 to station 3 after everyone arrives.

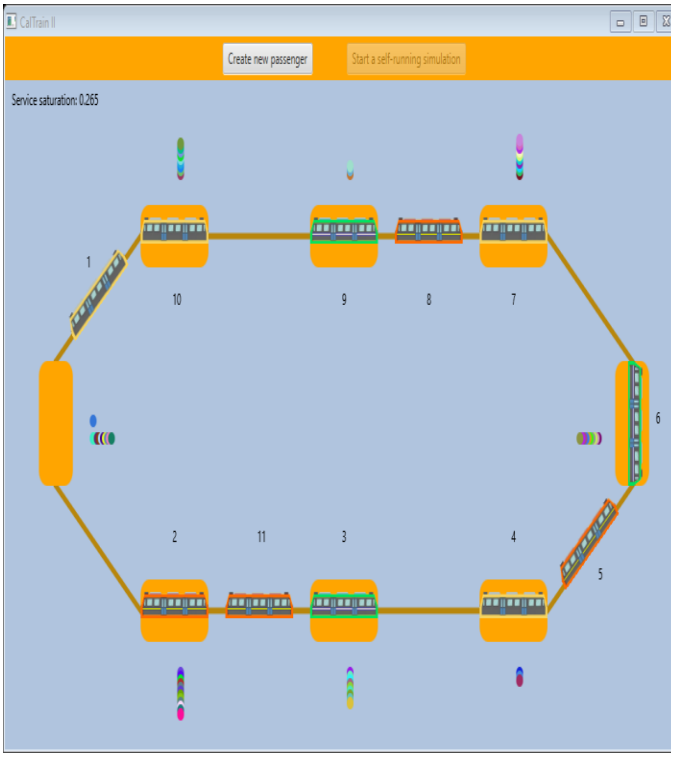
This test is designed to test how efficient the synchronization systems are with respect to groups of five passengers traversing the system with only one-passenger capacity trains.

1) **Monitors**
   a. **Roundtrip time:** 33.39 seconds
   b. **Roundtrip time:** 35.69 seconds
   c. **Roundtrip time:** 36.26 seconds
   d. **Average RTT: 35.13 seconds**
2) **Semaphores**
   a. **Roundtrip time:** 33.14 seconds
   b. **Roundtrip time:** 33.14 seconds
   c. **Roundtrip time:** 36.02 seconds
   d. **Average RTT: 34.10 seconds**

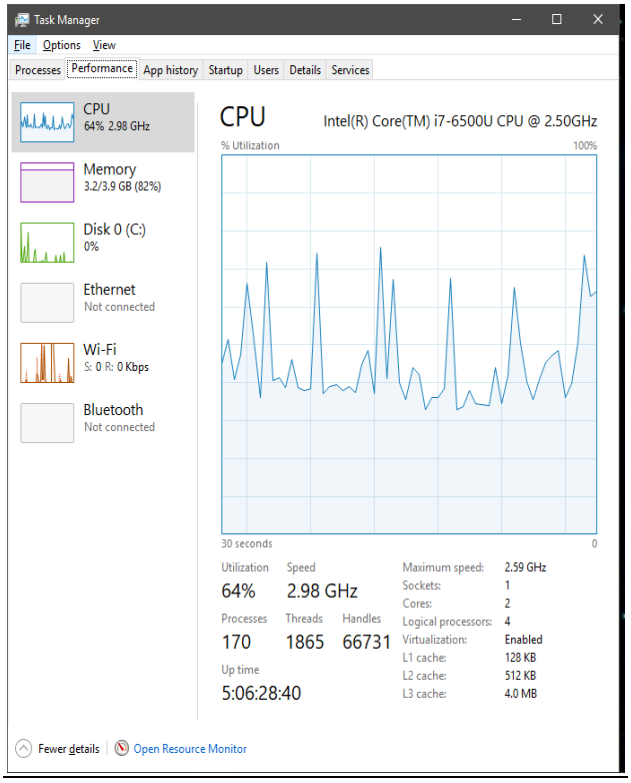
Sample Runs



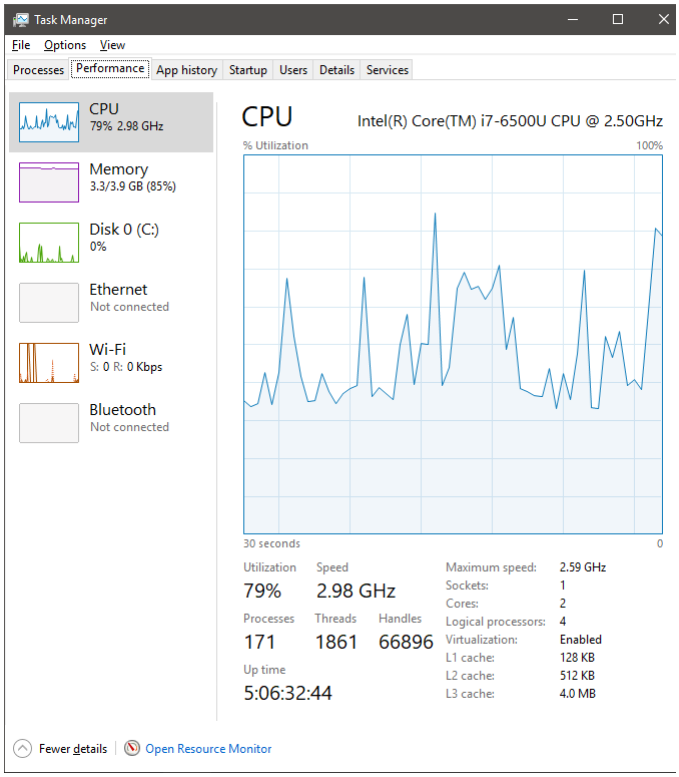




CPU utilization while running the program (using monitors)

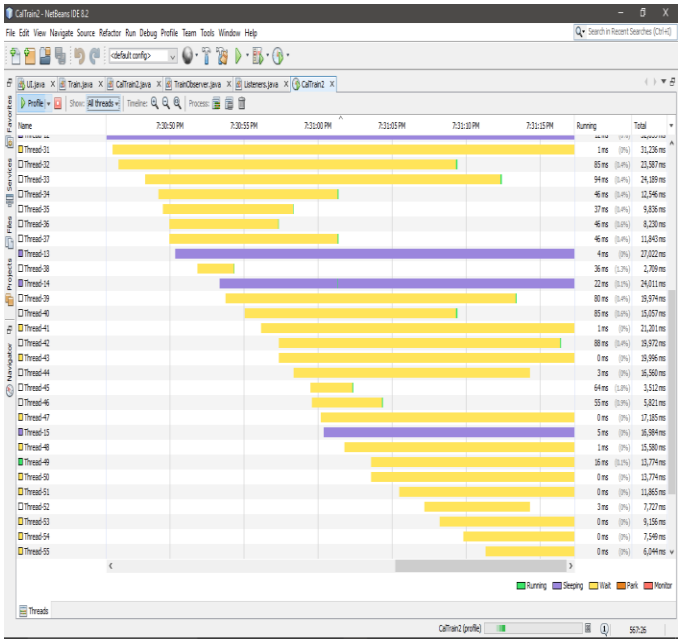


CPU utilization while running the program (using semaphores)

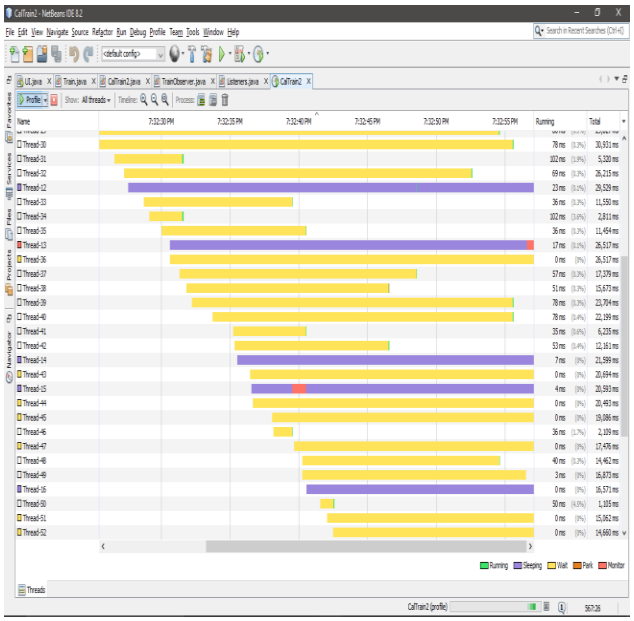


Note: Note the periodic spikes coinciding with the movement of trains.

Thread map of the program at 60 seconds (using monitors)



Thread map of the program at 60 seconds (using semaphores)



IV. Conclusion

A Posteriori Analysis

Rather unexpectedly, the results show little to no differences between the performances of semaphores and monitors. In fact, the results even favor semaphores slightly. Upon closer inspection, analysis, and study of the data collected by the developer-researchers, however, it appears that the Java implementation of semaphores differ from what was expected by the team as reflected in their a priori analysis.

According to the official Java documentation (Oracle, n.d.):

"If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until (...)"

When a thread is disabled, it does **not** use any CPU time *at all*. Hence, **a waiting semaphore will not consume any CPU time**, disproving the a priori analysis. Moreover, a thread can only be disabled using services related to an object's intrinsic lock. Hence, Java's semaphores use the same mechanisms as its internal monitors, if not the internal monitors themselves. Hence, **Java's semaphores are akin to abstracted versions of monitors.** The reason why the tests showed a minor delta towards semaphores in terms of performance may be because of Java's internal abstraction mechanisms which are obviously superior to the developers' naïve implementations of monitors and are specifically hand-optimized for performance. In the end, though, the differences are minor and statistically insignificant – a testament to the similarities of the Java semaphore and its internal monitors.

Acknowledgement

Our group wishes to thank Dr. Remedios Bulos for giving us the opportunity to work on this project in order to improve our know-how in computer process synchronization and for providing us with the guidance and feedback we need in order to complete the project. We can confidently say that we learned a lot in doing this project. Furthermore, we appreciate how the project gives us the opportunity to view real-life train systems from a distinct perspective, albeit from an artificial one. The country clearly needs a lot of knowledge on scaling down (or up) synchronized systems and how it impacts the community.

The following references served as valuable facets of information in order to lay the groundwork for our implementation of CalTrain II:

References

[1] The Philippine Star. 2016. Metro Manila traffic ranked among 10 worst in world. (October 2016). Retrieved July 19, 2017 from http://www.philstar.com/headlines/2016/10/20/1635484/metro-manila-traffic-ranked-among-10-worst-world

[2] Babe Romualdez. 2014. MRT's plague of problems. (August 2014). Retrieved July 19, 2017 from http://www.philstar.com/business/2014/08/21/1359803/mrts-plague-problems

[3] Oracle. n.d. Semaphore (Java Platform SE 7). (n.d.). Retrieved July 23, 2017 from https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html#acquire()

Index

Train.java

```
/*
 * To change this license header, choose
License Headers in Project Properties.
 * To change this template file, choose
Tools | Templates
 * and open the template in the editor.
 */
package Model.Core;

import Controller.CalTrain2;
import Model.Observer.TrainObserver;
import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author user
 */
public class Train implements Runnable {

    // Core attributes
    private final int num;
    private final int capacity;
    private int taken;
    private boolean isFull;
    private final String color;

    private final int[]
destinationCounters;

    // Link attributes
    private final FreeSegment entryPoint;
    private Segment currSegment;
    private Segment nextSegment;

    // Observers
    private final CalTrain2 dispatcher;
    private final TrainObserver
trainObserver;

    // Constructor
    public Train(int num, int capacity,
FreeSegment entryPoint, String color,
CalTrain2 dispatcher) {
        this.num = num;
        this.capacity = capacity;
        this.taken = 0;
        this.isFull = false;
        this.color = color;
        this.destinationCounters = new
int[CalTrain2.NUM_STATIONS];

        this.entryPoint = entryPoint;

        this.currSegment = entryPoint;
        this.nextSegment =
currSegment.getNextSegment();

        this.dispatcher = dispatcher;
        this.trainObserver = new
TrainObserver(this.dispatcher);
    }

    /**
     * @return the color
     */
    public String getColor() {
        return color;
    }

    /**
     * @return the num
     */
    public int getNum() {
        return num;
    }

    /**
     * @return the capacity
     */
    public int getCapacity() {
        return capacity;
    }

    /**
     * @return the taken
     */
    public int getTaken() {
        return taken;
    }

    /**
     * @param taken the taken to set
     */
    public void setTaken(int taken) {
        this.taken = taken;
    }

    /**
     * @return the isFull
     */
    public boolean isFull() {
        return isFull;
    }

    /**
```

```
     * @param isFull the isFull to set
     */
    public void setIsFull(boolean isFull)
{
        this.isFull = isFull;
    }

    /**
     * @return the destinationCounters
     */
    public int[] getDestinationCounters()
{
        return destinationCounters;
    }

    /**
     * @param destinationCounters the
destinationCounters to set
     */
    public void
setDestinationCounters(int[]
destinationCounters) {

this.setDestinationCounters(destinationCou
nters);
    }

    /**
     * @return the entryPoint
     */
    public FreeSegment getEntryPoint() {
        return entryPoint;
    }

    /**
     * @return the currSegment
     */
    public Segment getCurrSegment() {
        return currSegment;
    }

    /**
     * @param currSegment the currSegment
to set
     */
    public void setCurrSegment(Segment
currSegment) {
        this.currSegment = currSegment;
    }

    /**
     * @return the nextSegment
     */
    public Segment getNextSegment() {
        return nextSegment;
    }

    /**
     * @param nextSegment the nextSegment
to set
     */
    public void setNextSegment(Segment
nextSegment) {
        this.nextSegment = nextSegment;
    }

    // Move forward
    private void proceed() {
        // We've just moved to the next
segment
        currSegment = nextSegment;
        nextSegment =
currSegment.getNextSegment();
    }

    /**
     * @return the trainObserver
     */
    public TrainObserver
getTrainObserver() {
        return trainObserver;
    }

    /**
     * @return the dispatcher
     */
    public CalTrain2 getDispatcher() {
        return dispatcher;
    }

    // Alight passengers
    private void
alightPassengersMonitor(boolean showMsgs)
{
        //print("Train #" + num + " is
alighting passengers...", showMsgs);

        // This is the current station
monitor
        // Notify the passengers that
we're ready to take them out
        Station currentStationMonitor =
((Station)
currSegment).getStationAlightLock();

        synchronized
(currentStationMonitor) {

currentStationMonitor.notifyAll();
        }

        // A one second delay to make the
animation easier to follow
        try {
```

```
            Thread.sleep(1000);
        } catch (InterruptedException ex)
{
            trainObserver.print("Services
interrupted.", showMsgs);
        }
    }

    // Load passengers
    // station_load_train(struct station
*station, int count)
    private void
loadPassengersMonitor(boolean showMsgs) {
        //print("Train #" + num + " is
boarding passengers...", showMsgs);

        // This is the current station
monitor
        // Notify the passengers that
we're ready to take them in
        Station currentStationMonitor =
((Station)
currSegment).getStationBoardLock();

        synchronized
(currentStationMonitor) {

currentStationMonitor.notifyAll();
        }

        // A one second delay to make the
animation easier to follow
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex)
{
            trainObserver.print("Services
interrupted.", showMsgs);
        }
    }

    // Alight passengers
    private void
alightPassengersSemaphore(boolean
showMsgs) {
        //print("Train #" + num + " is
alighting passengers...", showMsgs);

        // If there are no passengers to
alight, never mind
        int alighting =
destinationCounters[((Station)
currSegment).getNum() / 2];

        // This is the current station
semaphore
        // Notify the passengers that
we're ready to take them out
        Semaphore currentStationSemaphore
= ((Station)
currSegment).getStationAlightSemaphore();


currentStationSemaphore.release(alighting)
;

        // A one second delay to make the
animation easier to follow
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex)
{
            trainObserver.print("Services
interrupted.", showMsgs);
        }
    }

    // Load passengers
    // station_load_train(struct station
*station, int count)
    private void
loadPassengersSemaphore(boolean showMsgs)
{
        //print("Train #" + num + " is
boarding passengers...", showMsgs);

        // If there are no passengers to
board or we're full, never mind
        int boarding = ((Station)
currSegment).getWaiting();

        // This is the current station
semaphore
        // Notify the passengers that
we're ready to take them in
        Semaphore currentStationSemaphore
= ((Station)
currSegment).getStationBoardSemaphore();

        // Hop on!

currentStationSemaphore.release(boarding);

        // A one second delay to make the
animation easier to follow
        try {
            Thread.sleep(1000);
        } catch (InterruptedException ex)
{
            trainObserver.print("Services
interrupted.", showMsgs);
        }
    }
```

```
    // Pause the thread for n seconds
    public void sleep(int n, boolean
showMsgs) {
        try {
            Thread.sleep(n);
        } catch (InterruptedException ex)
{
            trainObserver.print("Services
interrupted.", showMsgs);
        }
    }

    @Override
    public void run() {
        // Debug variable to use monitors
or not
        boolean useMonitors = true;

        // Debug variable to show print
statements (or not)
        boolean showMsgs = false;

        // Dispatch delay
        sleep(2000, showMsgs);

        // Use monitors or not?
        if (useMonitors) {
            runWithMonitors(showMsgs);
        } else {
            runWithSemaphores(showMsgs);
        }
    }

    // Run with monitors
    public void runWithMonitors(boolean
showMsgs) {
        // This is the entry monitor
        Segment entryPointMonitor =
entryPoint;

        synchronized (entryPointMonitor) {
            try {
                // Wait until the entry
point branch is clear
                // This loop guards
against spurious wakeups as recommended
                // by the official Java
documentation
                while
(entryPoint.isOccupied()) {

entryPointMonitor.wait();
                }

                // Take this spot

entryPoint.setIsOccupied(true);

entryPoint.setTrainInside(this);

                // The next segment
monitor
                Segment nextSegmentMonitor
= nextSegment;

                synchronized
(nextSegmentMonitor) {
                    // Wait until the
segment AFTER the entry point branch is
unreserved
                    // This loop guards
against spurious wakeups as recommended
                    // by the official
Java documentation
                    while
(nextSegment.isReserved()) {

nextSegmentMonitor.wait();
                    }

                    // Immediately claim
the next segment

nextSegment.setIsReserved(true);
                }

                // Leave this spot

entryPoint.setIsOccupied(false);

entryPoint.setTrainInside(null);
            } catch (InterruptedException
ex) {

trainObserver.print("Services
interrupted.", showMsgs);
            }

            // Enter the line!
            proceed();

            // Notify the trains that are
waiting for this segment to be clear
            entryPointMonitor.notify();
        }

        // Run this code indefinitely
        while (true) {
            // This is the current segment
monitor
            // Only one train can ever
occupy this segment
```

```
            // Take note of the current
segment
            Segment currSegmentMonitor =
currSegment;

            synchronized
(currSegmentMonitor) {
                // Take this spot

currSegment.setIsOccupied(true);

currSegment.setTrainInside(this);

currSegment.setIsReserved(false);

                // Notify this train's
observer that its position has changed
                // Console position update

trainObserver.update(showMsgs);

                // GUI position update

trainObserver.updateTrainPosition(this,
currSegment);

                // If this segment is a
station, load and unload passengers
                if (currSegment instanceof
Station) {
                    // Open doors and
allow passengers to get off and on

alightPassengersMonitor(showMsgs);

loadPassengersMonitor(showMsgs);

                    // Check if we need
more trains!
                    requestTrain();
                } else {
                    // Else, take a second
to move
                    sleep(1000, showMsgs);
                }

                // This is the next
segment monitor
                Segment nextSegmentMonitor
= nextSegment;

                synchronized
(nextSegmentMonitor) {
                    // Is it okay to
proceed?
                    try {
                        // Wait until the
next segment is clear
                        // This loop
guards against spurious wakeups as
recommended
                        // by the official
Java documentation
                        while
(nextSegment.isReserved()) {

nextSegmentMonitor.wait();
                        }

                        // Immediately
claim the next segment

nextSegment.setIsReserved(true);
                    } catch
(InterruptedException ex) {

trainObserver.print("Services
interrupted.", showMsgs);
                    }
                }

                // Leave this spot

currSegment.setIsOccupied(false);

currSegment.setTrainInside(null);

                // If ready, then proceed
                proceed();

                // Then tell others we're
done occupying that spot

currSegmentMonitor.notify();
            }
        }
    }

    // Run with semaphores
    public void runWithSemaphores(boolean
showMsgs) {
        try {
            // This is the entry point
semaphore
            Semaphore entryPointSemaphore
= entryPoint.getSemaphore();

            // Wait until the entry point
branch is clear
            entryPointSemaphore.acquire();

            // Take this spot
```

```
entryPoint.setIsOccupied(true);

entryPoint.setTrainInside(this);

            // This is the entry next
segment semaphore
            Semaphore
entryNextSegmentSemaphore =
nextSegment.getSemaphore();

            // Wait until the segment
AFTER the entry point branch is unreserved

entryNextSegmentSemaphore.acquire();

            // Immediately claim the next
segment

nextSegment.setIsReserved(true);

            // Allow trains already in the
loop to continue

entryNextSegmentSemaphore.release();

            // Leave this spot

entryPoint.setIsOccupied(false);

entryPoint.setTrainInside(null);

            // Enter the line!
            proceed();

            // Notify the trains that are
waiting for this segment to be clear
            entryPointSemaphore.release();

            // Run this code indefinitely
            while (true) {
                // This is the current
segment semaphore
                // Only one train can ever
occupy this segment
                // Take note of the
current segment
                Semaphore
currentSegmentSemaphore =
currSegment.getSemaphore();

                // Lock this segment

currentSegmentSemaphore.acquire();

                // Take this spot

currSegment.setIsOccupied(true);

currSegment.setTrainInside(this);

currSegment.setIsReserved(false);

                // Notify this train's
observer that its position has changed
                // Console position update

trainObserver.update(showMsgs);

                // GUI position update

trainObserver.updateTrainPosition(this,
currSegment);

                // If this segment is a
station, load and unload passengers
                if (currSegment instanceof
Station) {
                    // Open doors and
allow passengers to get off and on

alightPassengersSemaphore(showMsgs);

loadPassengersSemaphore(showMsgs);

                    // Check if we need
more trains
                    requestTrain();
                } else {
                    // Else, take a second
to move
                    sleep(1000, showMsgs);
                }

                // This is the next
segment semaphore
                Semaphore
nextSegmentSemaphore =
nextSegment.getSemaphore();

                // Is it okay to proceed?
                // Wait until the next
segment is clear

nextSegmentSemaphore.acquire();

                // Immediately claim the
next segment

nextSegment.setIsReserved(true);

                // Move forward
```

```
nextSegmentSemaphore.release();

                // Leave this spot

currSegment.setIsOccupied(false);

currSegment.setTrainInside(null);

                // If ready, then proceed
                proceed();

                // Then tell others we're
done occupying that spot

currentSegmentSemaphore.release();
            }
        } catch (InterruptedException ex)
{
            System.out.println("Services
interrupted.");
        }
    }

    // Reserve a seat
    public synchronized boolean
reserveSeat() {
        System.out.print("Train " + num +
": " + taken + " to ");
        if (taken < capacity) {
            taken++;

            if (taken == capacity) {
                isFull = true;
            }

            return true;
        } else {
            return false;
        }
    }

    // Leave a seat
    public synchronized void leaveSeat() {
        taken--;

        isFull = false;
    }

    // Request train heuristic
    public synchronized void
requestTrain() {
        trainObserver.demandTrain();
    }

    @Override
    public boolean equals(Object object) {
        if (object == null) {
            return false;
        }

        return ((Train) object).num ==
this.num;
    }

    @Override
    public int hashCode() {
        int hash = 7;

        hash = 67 * hash + this.num;

        return hash;
    }
}
```

Passenger.*java*

```
/*
 * To change this license header, choose
License Headers in Project Properties.
 * To change this template file, choose
Tools | Templates
 * and open the template in the editor.
 */
package Model.Core;

import Controller.CalTrain2;
import Model.Observer.PassengerObserver;
import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import java.util.logging.Logger;
import javafx.scene.paint.Color;

/**
 *
 * @author user
 */
public class Passenger implements Runnable
{

    // Core attributes
    private final String name;
    private Station origin;
    private Station destination;
    private Status status;
    private Train currTrain;
    private Color color;
    private final boolean isTracked;

    // Observers
    private final PassengerObserver
passengerObserver;
```

```
    public enum Status {
        WAITING, BOARDING, SEATED,
ALIGHTING, ARRIVED
    };

    // Constructor
    public Passenger(String name, int
origin, int destination, boolean
isTracked, CalTrain2 dispatcher) {
        this.name = name;
        this.origin = (Station)
dispatcher.getSegments()[origin * 2 - 1];
        this.destination = (Station)
dispatcher.getSegments()[destination * 2 -
1];
        this.status = Status.WAITING;
        this.currTrain = null;
        this.color =
Color.color(Math.random(), Math.random(),
Math.random());
        this.isTracked = isTracked;

        this.passengerObserver = new
PassengerObserver(dispatcher);
    }

    /**
     * @return the name
     */
    public String getName() {
        return name;
    }

    /**
     * @return the origin
     */
    public Station getOrigin() {
        return origin;
    }

    /**
     * @param origin the origin to set
     */
    public void setOrigin(Station origin)
{
        this.origin = origin;
    }

    /**
     * @return the destination
     */
    public Station getDestination() {
        return destination;
    }

    /**
     * @param destination the destination
to set
     */
    public void setDestination(Station
destination) {
        this.destination = destination;
    }

    /**
     * @return the status
     */
    public Status getStatus() {
        return status;
    }

    /**
     * @param status the status to set
     */
    public void setStatus(Status status) {
        this.status = status;
    }

    /**
     * @return the color
     */
    public Color getColor() {
        return color;
    }

    /**
     * @param color the color to set
     */
    public void setColor(Color color) {
        this.color = color;
    }

    /**
     * @return the currTrain
     */
    public Train getCurrTrain() {
        return currTrain;
    }

    /**
     * @param currTrain the currTrain to
set
     */
    public void setCurrTrain(Train
currTrain) {
        this.currTrain = currTrain;
    }

    /**
     * @return the isTracked
     */
    public boolean isTracked() {
        return isTracked;
```

```
    }

    @Override
    public void run() {
        // Debug variable to use monitors
or not
        boolean useMonitors = true;

        // Debug variable to show print
statements (or not)
        boolean showMsgs = true;

        // Use monitors or not?
        if (useMonitors) {
            runWithMonitors(showMsgs);
        } else {
            runWithSemaphores(showMsgs);
        }
    }

    // Run with monitors
    public void runWithMonitors(boolean
showMsgs) {
        // Is this passenger in his/her
destination station?
        boolean isArrived = false;

        // Run this code indefinitely
        while (!isArrived) {
            // What the passenger does
depends on his/her status
            switch (status) {
                case WAITING:
                    // Hey look, a
passenger!

passengerObserver.addPassenger();

                    // Wait for the train
                    // This is the origin
station monitor
                    Station
originStationMonitor =
origin.getStationBoardLock();

                    //
station_wait_for_train(struct station
*station)
                    synchronized
(originStationMonitor) {
                        try {
                            // Console
position update

passengerObserver.print("Passenger " +
name + " is waiting at " +
origin.getName(), showMsgs);

                            // GUI
position update (waiting)

passengerObserver.updatePassengerPosition(
this, origin);

                            // Tracker
position update
                            if (isTracked)
{

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                            }

                            // Wait until
the train is ready to pick the
                            // passengers
up or until a train with free
                            // seats is
available
                            // This loop
guards against spurious wakeups as
recommended
                            // by the
official Java documentation
                            while
(!origin.isOccupied() ||
!origin.getTrainInside().reserveSeat()) {

originStationMonitor.wait();
                            }
                        } catch
(InterruptedException ex) {

passengerObserver.print("Services
interrupted.", showMsgs);
                        }
                    }

                    // Get ready to board!
                    status =
Status.BOARDING;

                    break;
                case BOARDING:
                    // This is the time
between the passengers waiting for the
                    // train and actually
sitting down
                    // i.e., the
passengers are in the process of boarding

                    // This passenger is
now getting in the train
```

```
                    currTrain =
origin.getTrainInside();

                    // Console position
update

passengerObserver.print("Passenger " +
name + " is boarding Train " +
currTrain.getNum(), showMsgs);

                    // GUI position update
(boarding)

passengerObserver.updatePassengerPosition(
this, origin);

                    // Tracker position
update
                    if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                    }

                    // Just sit down
                    status =
Status.SEATED;

                    break;
                case SEATED:
                    // Wait for the
destination station
                    // This is the
destination monitor
                    Station
destinationStationMonitor =
destination.getStationAlightLock();

                    // Tracker position
update
                    if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                    }

                    synchronized
(destinationStationMonitor) {
                        try {
                            // Console
position update

passengerObserver.print("Passenger " +
name + " is waiting for "
                                    +
destination.getName() + " at Train " +
currTrain.getNum(), showMsgs);

                            // Wait until
the destination station is occupied
                            // and the
train occupying it is the very train this
                            // passenger
is on
                            // This loop
guards against spurious wakeups as
recommended
                            // by the
official Java documentation
                            while
(!destination.isOccupied() ||
!destination.getTrainInside().equals(currT
rain)) {

destinationStationMonitor.wait();
                            }
                        } catch
(InterruptedException ex) {

passengerObserver.print("Services
interrupted.", showMsgs);
                            }
                    }

                    // If it is this
station, get off!
                    status =
Status.ALIGHTING;

                    break;
                case ALIGHTING:
                    // Console position
update

passengerObserver.print("Passenger " +
name + " is alighting.", showMsgs);

                    // GUI position update
(alighting)

passengerObserver.updatePassengerPosition(
this, destination);

                    // Tracker position
update
                    if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, destination, status);
                    }

                    // Leave the seat
                    currTrain.leaveSeat();
```

```
                    // This passenger is
now getting off the train
                    currTrain = null;

                    // We've arrived!
                    status =
Status.ARRIVED;

                    break;
                case ARRIVED:
                    // Console position
update

passengerObserver.print("Passenger " +
name + " is done!", showMsgs);

                    // GUI position update
(arrived)

passengerObserver.updatePassengerPosition(
this, destination);

                    // Tracker position
update
                    if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, destination, status);
                    }

                    // Bye passenger!

passengerObserver.removePassenger();

                    // Get out because
we're here and we're done!
                    isArrived = true;

                    break;
            }
        }
    }

    // Run with semaphores
    public void runWithSemaphores(boolean
showMsgs) {
        try {
            // Is this passenger in
his/her destination station?
            boolean isArrived = false;

            // Run this code indefinitely
            while (!isArrived) {
                // What the passenger does
depends on his/her status
                switch (status) {
                    case WAITING:
                        // Hey look, a
passenger!

passengerObserver.addPassenger();

                        // One more
passenger is now waiting

//origin.addWaitingPassenger();
                        // Wait for the
train
                        //
station_wait_for_train(struct station
*station)
                        // This is the
origin station semaphore
                        Semaphore
originStationBoardSemaphore =
origin.getStationBoardSemaphore();

                        // Console
position update

passengerObserver.print("Passenger " +
name + " is waiting at " +
origin.getName(), showMsgs);

                        // GUI position
update (waiting)

passengerObserver.updatePassengerPosition(
this, origin);

                        // Tracker
position update
                        if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                        }

                        // Wait until the
train is ready to pick the
                        // passengers up
or until a train with free
                        // seats is
available
                        while
(!origin.isOccupied() ||
!origin.getTrainInside().reserveSeat()) {

originStationBoardSemaphore.acquire();
                        }

                        // Get ready to
board!
```

```
                        status =
Status.BOARDING;

                        break;
                    case BOARDING:
                        // This is the
time between the passengers waiting for
the
                        // train and
actually sitting down
                        // i.e., the
passengers are in the process of boarding
                        // This passenger
is now getting in the train
                        currTrain =
origin.getTrainInside();

                        // Console
position update

passengerObserver.print("Passenger " +
name + " is boarding Train " +
currTrain.getNum(), showMsgs);

                        // GUI position
update (boarding)

passengerObserver.updatePassengerPosition(
this, origin);

                        // Tracker
position update
                        if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                        }

                        // Remove a
waiting passenger

//origin.removeWaitingPassenger();
                        // Just sit down
                        status =
Status.SEATED;

                        break;
                    case SEATED:
                        // Tracker
position update
                        if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, origin, status);
                        }

                        // Console
position update

passengerObserver.print("Passenger " +
name + " is waiting for "
                                    +
destination.getName() + " at Train " +
currTrain.getNum(), showMsgs);

                        // Wait for the
destination station
                        // This is the
destination semaphore
                        Semaphore
destinationStationSemaphore =
destination.getStationAlightSemaphore();

                        // Set your eyes
at the destination
                        while
(!destination.isOccupied() ||
!destination.getTrainInside().equals(currT
rain)) {

destinationStationSemaphore.acquire();
                        }

                        // If it is this
station, get off!
                        status =
Status.ALIGHTING;

                        break;
                    case ALIGHTING:
                        // Console
position update

passengerObserver.print("Passenger " +
name + " is alighting.", showMsgs);

                        // GUI position
update (alighting)

passengerObserver.updatePassengerPosition(
this, destination);

                        // Tracker
position update
                        if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, destination, status);
                        }

                        // Leave the seat

currTrain.leaveSeat();
```

```
                        // This passenger
is now getting off the train
                        currTrain = null;

                        // We've arrived!
                        status =
Status.ARRIVED;

                        break;
                    case ARRIVED:
                        // Console
position update

passengerObserver.print("Passenger " +
name + " is done!", showMsgs);

                        // GUI position
update (arrived)

passengerObserver.updatePassengerPosition(
this, destination);

                        // Tracker
position update
                        if (isTracked) {

passengerObserver.updatePassengerTrack(thi
s, destination, status);
                        }

                        // Bye passenger!

passengerObserver.removePassenger();

                        // Get out because
we're here and we're done!
                        isArrived = true;

                        break;
                }
            }
        } catch (InterruptedException ex)
{

passengerObserver.print("Services
interrupted.", showMsgs);
        }
    }
}
```